Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Škerjanc

Avtomatsko iskanje števcev objektov na slikah

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Luka Šajn

Ljubljana, 2016

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Anže Škerjanc sem avtor magistrskega dela z naslovom:

Avtomatsko iskanje števcev objektov na slikah

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom doc. dr. Luka Šajna,
- so elektronska oblika magistrskega dela, naslov (slovenski, angleški), povzetek (slovenski, angleški) ter ključne besede (slovenske, angleške) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 3. aprila 2016 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Luku Šajnu za vso strokovno pomoč, nasvete, mentorstvo in vodenje pri izdelavi in pisanju magistrske naloge. Poleg tega gre zahvala vsem, ki so prispevali testne domene slik.

Prav tako gre zahvala vsem domačim, ki so me podpirali, spodbujali in verjeli vame v času mojega študija.

Kazalo

Povzetek

Ročno štetje objektov na slikah je časovno zamudno opravilo. Zato si lahko pomagamo z algoritmi, ki avtomatizirajo štetje objektov. Taki algoritmi vsebujejo veliko nastavitev. Nastavitve algoritmov, konstant in rutin zamenjamo s parametri. Vrednost parametrov je treba za optimalno delovanje nad posamezno domeno slik optimizirati. Ročno nastavljanje je težavno, saj parametri med seboj niso neodvisni. Naš cilj je poiskati optimizacijski algoritem, ki bo iz podanega opisa algoritma za štetje in ustrezne domene, v katerem naj bi se ta algoritem uporabljal, poiskal nabor parametrov, ki najbolje štejejo iskane objekte. Zaradi velikega obsega možnih kombinacij (kartezijski produkt domen parametrov) celotni pregled prostora parametrov ni mogoč, zato smo uporabili genetski algoritem. Končni produkt je orodje, ki bo preprosto in uporabno za strokovnjake brez kakršnega koli tehničnega znanja in vedenja o konceptih, ki se skrivajo za temi algoritmi.

Ključne besede

štetje objektov, optimizacija parametrov, genetski algoritem, ImageJ, makro jezik, števec

Abstract

Manual counting of objects on images is a time-consuming task, therefore we can make us of algorithms that automate object counting. Such algorithms have many settings. We substitute the settings of algorithms, constants and routines with parameters. The value of parameters for individual images has to be optimized for optimal functioning. Manual settings are hard to handle, since the parameters are interconnected. Our goal is to find an optimization algorithm, which will find a set of most suitable parameters for object counting out of the given description of the counting algorithm and the suitable domain, where this algorithm is going to be used. Due to a large number of possible complications (Cartesian product of domains of parameters) a complete overview of the space of parameters in impossible, therefore we used a genetic algorithm. The final product is a tool, which will be simple and useful for experts without any technical knowledge about concepts, hiding behind these algorithms.

Keywords

object counting, optimization of parameters, genetic algorithm, ImageJ, macro language, counter

Poglavje 1

Uvod

Pri reševanju problemov je velikokrat potrebna ocena števila objektov na sliki. Iskani objekti so lahko na primer celice, ptice, mikroorganizmi itd. Veliko takega štetja se še vedno opravlja ročno, kar je zamudno in naporno delo. Z računalnikom se lahko samodejno prešteje velik del takih objektov. Težava je v tem, ker se s takimi problemi pogosto srečujejo ljudje (večinoma znanstveniki), ki niso računalniško dovolj spretni, da bi si štetje avtomatizirali, in so odvisni od zmogljivosti plačljivih orodij. Če ta orodja odpovedo na neki domeni slik, jim preostane le še ročno štetje objektov.

Veliko algoritmov se je že razvilo za reševanje problema štetja. Ti algoritmi večinoma vsebujejo veliko parametrov, ki jih je treba nastaviti. Tako se lahko prilagodijo različnim domenam slik. Nastaviti te parametre algoritma ni lahko opravilo. Poleg tega ni zagotovila, da se bo našla optimalna konfiguracija nastavitev. Človek tako nastavljanje opravlja s tako imenovano požrešno metodo. Požrešno nastavljanje pomeni, da se parametri nastavljajo neodvisno drug za drugim. Da s požrešno metodo dobimo tudi najbolj optimalno rešitev, mora veljati popolna neodvisnost parametrov algoritma. Ob tem moramo učinke nastavljanja opazovati na celotni množici testnih slik, kar za človeka ni lahko opravilo [1].

Zato smo si zadali nalogo, da poskusimo parametre nastaviti samodejno. Iz algoritma bomo pridobili vse parametre, ki jih ta vsebuje. Uporabili bomo

algoritme, ki so spisani v makro jeziku ImageJ. ImageJ je orodje za obdelavo in analizo slik. Da bi lahko izvedli optimizacijo, smo morali makro jezik razširiti. Algoritem za štetje ima hitro lahko veliko parametrov, s tem postane prostor vseh možnih stanj ogromen. Preiskati celotni prostor stanj zato ne pride v poštev. Nastavljanje bomo zato prepustili genetskemu algoritmu. Hevristika genetskega algoritma bo učna množica slik, ki bo prešteta ročno. V članku »Automatically Searching for Optimal Paramter Settings Using a Genetic Algorithm« [2] so navedli, da lahko pričakujemo boljše rezultate algoritma, katerega parametre smo optimizirali z genetskim algoritmom. To so pokazali na algoritmu za prepoznavo obrazov, pri čemer so z optimizacijo nastavitev dosegli boljše rezultate kot avtorji algoritma.

Uporabo našega algoritma smo želeli narediti čim preprostejšo. Od znanstvenika se tako zahteva le osnovno poznavanje makro jezika ImageJ. Zato smo morali dodajanje razširitev v makro jezik narediti čim preprostejše in smo za delo razvili preprost uporabniški vmesnik.

Uspešnost našega algoritma smo preizkusili na stvarnih problemih, ki smo jih dobili iz različnih virov. V primeru naše uspešnosti štetja večino zanima tudi nadaljnje sodelovanje z nami, saj imajo še domene, ki bi jih želeli znati šteti.

Poglavje 2

ImageJ

ImageJ je preprosto orodje za obdelavo in analizo slik, katerega izvorna koda je prosto dostopna. Kljub njegovi preprostosti ostaja močno orodje za obdelavo slik. Napisan je v Javi, kar mu omogoča poganjanje na katerem koli računalniku s podporo javanskemu navideznemu mehanizmu. ImageJ lahko uporabljamo kot samostojno aplikacijo ali kot programsko knjižnico za delo s slikami [3].

Arhitektura ImageJ je odprta, kar pomeni, da omogoča veliko razširljivost. Razširljivost je omogočena z vtičniki, ki so napisani v Javi. Programiranje v Javi odpre veliko možnosti, ki jih lahko naredimo v vtičniku. Java se izkaže kot dober programski jezik za pisanje razširitev. Je razširjen jezik, z veliko že spisanih funkcionalnosti. Je tudi dober kompromis med hitrostjo izvajanja in preprostostjo integracije vtičnika v orodje ImageJ. Spisana razširitev je prenosljiva med sistemi. Na spletu se najde veliko spisanih razširitev, ki so prosto dostopne [4].

2.1 Makro jezik ImageJ

ImageJ ima vgrajeno podporo makrojev. Makro je skriptni program, s katerim izvedemo zaporedje operacij. S tem poenostavimo izvajanje ponavljajočega se zaporedja operacij nad slikami. Dobri lastnosti makro jezika

sta njegovi preprostost in uporabnost. Operacije, izvedene na uporabniškem vmesniku, se lahko izvedejo v makroju. Jezik vsebuje [3]:

1. ***spremenljivke:*** uporablja netipizirane spremenljivke, katerih tip se nastavi ob inicializaciji. Nekaj primerov deklaracije in inicializacije spremenljivk:

$$v = 1; \leftarrow \check{s}tevilo$$
$$v = \text{"}niz\text{"}; \leftarrow niz$$
$$v = newArray(13, 44, 16); \leftarrow tabela;$$

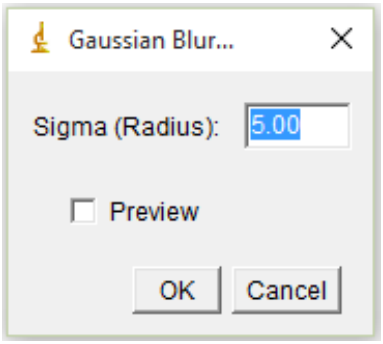
2. ***operatorje:*** podprtih je večina operatorjev, ki jih pozna Java, na primer vsota, razlika, množenje, deljenje, negacija, logični IN, logični ALI, enakost, večje, manjše itd.;

3. ***pogojne stavke:*** pozna pogojne stavke »if-else«. Sintaksa in obnašanje sta enaka javanskima. Primer sintakse pogojnega stavka:

$$\begin{aligned} &if\ (pogoj)\{ \\ &\quad stavki \\ &\}\ else\ \{ \\ &\quad stavki \\ &\}; \end{aligned}$$

4. ***zanke:*** pozna osnovne zanke, kot so for, while in do-while:

$$\begin{aligned} &for\ (start;\ pogoj;\ korak)\ \{ \\ &\quad stavki \\ &\} \\ &while\ (pogoj)\ \{ \\ &\quad stavki \\ &\} \end{aligned}$$



Slika 2.1: Dialog za poganjanje in nastavljanje ukaza »Gaussian Blur...«

5. ***funkcije:*** vsebuje veliko vgrajenih funkcij. Sintaksa funkcije je:

$$ime\_funckije(param_1,\ param_2,\ \cdots,\ param_N);$$

Najbolj uporabna je funkcija »run«. Ta funkcija požene ukaz orodja ImageJ. Prvi parameter funkcije je ime ukaza, ki ga želimo izvesti, drugi parameter pa so nastavitve ukaza. Na sliki 2.1 je na primer prikazan dialog, v katerem nastavimo in poženemo ukaz »Gaussian Blur...«. Enak učinek lahko naredimo s funkcijo »run« na naslednji način:

$$run('Gaussian\ Blur...',\ 'sigma = 5');$$

Da je pisanje makrojev še lažje, je v orodju ImageJ na voljo funkcionalnost snemanja ukazov. Vse, kar naredimo prek uporabniškega vmesnika, se v ozadju zapisuje v makro. Tako pisanje makrojev postane preprosto opravilo tudi računalniško ne najbolj spretnim uporabnikom.

Pri orodju ImageJ nas je zmotila slaba implementacija interpreterja. Interpreter skrbi za izvajanje makrojev in je preveč vezan na uporabo v njihovem uporabniškem vmesniku, saj so nekatere nastavitve v okviru interpreterja globalne. Težava nastane pri paralelizaciji izvajanja v enem procesu (z nitmi). Zato je preostala edina možna paralelizacija na stopnji procesa, ki je pri implementaciji povzročala več težav. Poleg tega rešitev ni tako optimalna in elegantna.

Poglavje 3

Razširjeni makro jezik ImageJ

Za potrebe optimizacije števca se je pokazala potreba po razširitvi osnovnega makro jezika ImageJ. Z razširitvijo moramo v algoritmu števca omogočiti definicijo parametrov. Te parametre bo genetski algoritem pozneje poskušal optimizirati. Treba je še opredeliti način, kako bomo v aplikaciji pridobili objekte, ki jih je števec našel. S tem bomo dobili možnost ugotoviti kakovost števca.

Želeli smo, da bi naše razširitve jezika čim manj posegale v obstoječi makro. Tako bi bilo dodajanje razširitev preprostejše. Za osnovo sintakse našega razširjenega makro jezika smo uporabili označevalni jezik XML (»Extensible Markup Language«). XML se nam je zdel dober zaradi njegove razširjenosti: veliko ljudi zna delati z njim, ima dobro podporo razčlenjevanja, človeku je razumljiva in preprosta oblika zapisa strukturiranih podatkov, ob tem je prijazna za branje in pisanje. V algoritmu 1 sta prikazani dve možni sintaksi razširitve. Vsaka razširitev predpisuje obvezni atribut »ID«. To je enolični identifikator razširitve.

Razširitve jezika lahko razdelimo v skupine:

- parametrov,
- števcev,
- kontrolnih ukazov.

Algoritem 1: Sintaksa razširjenega jezika

```
1: //ukazi, ki ne vsebujejo vgnezdene kode
2: <ImeUkaza ID="enoličnaOznaka" parameter1="vrednost1"/>
3: //ukazi, ki vsebujejo vgnezdeno kodo
4: <ImeUkaza ID="enoličnaOznaka" parameter1="vrednost1">
5:     ukazi
6: </ImeUkaza>
```

3.1 Parametri

Konstantne vrednosti v algoritmih, ki jih želimo optimizirati, nadomestimo s parametri. Za te parametre bomo v fazi optimizacije poskušali poiskati njihove optimalne vrednosti. Vsakemu parametru lahko z atributom »value« nastavimo njegovo vrednost, ki jo bo vrnil v času izvajanja algoritma. Za potrebe optimizacije lahko določimo tudi njegovo najmanjšo in največjo dovoljeno vrednost parametra. Algoritem bo optimalno vrednost iskal v okviru teh omejitev. Z omejevanjem dovoljenih vrednosti lahko ogromno pridobimo glede zmanjšanja prostora stanj preiskovanja in posledično s tem precej pohitrimo postopek optimizacije števca.

Trenutno podprti tipi parametrov so:

- celo število (»ParameterInt«, tabela 3.1),
- decimalno število (»ParameterDouble«, tabela 3.2),
- logična vrednost (»ParameterBool«, tabela 3.3).

Tabela 3.1: Argumenti parametra tipa celo število

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
Value	NE	0	Vrednost parametra
MinValue	NE	/	Najmanjša dovoljena vrednost
MaxValue	NE	/	Največja dovoljena vrednost

Tabela 3.2: Argumenti parametra tipa decimalno število

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
Value	NE	0,0	Vrednost parametra
MinValue	NE	/	Najmanjša dovoljena vrednost
MaxValue	NE	/	Največja dovoljena vrednost

Tabela 3.3: Argumenti parametra tipa logična vrednost

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
Value	NE	false	Vrednost parametra

Algoritem 2 prikazuje preprost primer parametrizacije kode. Funkcija »setThreshold« v osnovni različici kot drugi parameter dobi fiksno vrednost 150. Pri parametrizaciji pa vrednost 150 zamenjamo s celoštevilskim parametrom. Ker smo vrednost parametra nastavili na 150, bo ob morebitnem poganjanju rezultat algoritma popolnoma enak. Parameter pa dobi svoj pomen pri optimizaciji, pri kateri bomo v okviru dovoljenega intervala vrednosti (med najmanjšo in največjo vrednostjo) poskusili poiskati najboljšo vrednost. S tem bomo nastavili vrednost parametra na optimalno vrednost, ki se bo uporabila pri izvajanju.

Algoritem 2: Parametrizacija kode ImageJ

```
1: //koda v jeziku ImageJ
2: setThreshold(0, 150)
3: //koda v razširjenem jeziku ImageJ
4: setThreshold(0, <ParameterInt minValue="0" maxValue="255" value="150" />)
```

Naš genetski algoritem izvaja optimizacijo edino nad parametri. Zato moramo vse optimizacije preostalih gradnikov prevesti na problem iskanja optimalnih vrednosti parametrov. Tako sta omogočena lažje dodajanje novih gradnikov in lažji potek optimizacije.

3.2 Števci

Da bomo makro lahko obravnavali kot števec objektov, ga moramo v celoti oviti v gradnik tipa števec. Naloga števca je iz obdelane slike vrniti koordinate objektov, ki jih je ta preštel. Implementirali in preizkusili smo števca »števec delcev« in »števec delcev z rastjo regij«. Menimo, da bosta ta dva števca dovolj za obravnavo večine problemov štetja, ki jih je mogoče rešiti z našim pristopom. Sintaksa in implementacija razširjenega jezika ImageJ dopuščata preprosto dodajanje novih gradnikov. Zato tega, če bi se izkazala potreba po novem števcu, ne bi bilo težko dodati v obstoječo funkcionalnost razširjenega jezika.

3.2.1 Števec delcev

Za osnovo implementacije smo uporabili že vgrajen ukaz »Analyze Particles...«. Ukaz prešteje in izmeri objekte v binarni sliki. Objekti, ki jih želimo prešteti, morajo imeti logično vrednost 1, ozadje pa 0. Ukaz (»ObjectCounterAnalyzeParticles«) vsebuje nastavitve, prikazane v tabeli 3.4. Mogoče je omejiti velikost in kompaktnost objektov, ki jih algoritem prešteje. Velikost objektov je enaka številu slikovnih pik, ki jih ta vsebuje. Kompaktnost (3.1) opisuje osnovno obliko objekta. Za najbolj kompakten lik velja krog, za katerega je kompaktnost enaka 1. Kompaktnost neskončno podolgovatega poligona pa je 0 [3].

$$kompaktnost = 4\pi \cdot \frac{velikost}{obseg^2} \tag{3.1}$$

Najmanjše in največje vrednosti nastavitev za omejitev velikosti in kompaktnosti v fazi optimizacije pretvorimo v parametre. Tako poskrbimo, da bo genetski algoritem optimiziral tudi te.

Tabela 3.4: Parametri števca.

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
MinSize	NE	0	Omejitev najmanjše velikosti
MaxSize	NE	100000	Omejitev največje velikosti
MinCirc	NE	0	Omejitev najmanjše kompaktnosti
MaxCirc	NE	1,0	Omejitev največje kompaktnosti

3.2.2 Števec delcev z rastjo področij

Zamisel za števec smo dobili na podlagi algoritma rasti področij iz semen (»seeded region growing«) [5]. Algoritem se uporablja za segmentacijo slik. Algoritem na vhodu pričakuje sliko z izbranimi semeni, izbire semen pa ne predpisuje. Izbira je prepuščena uporabniku ali morebiti samodejni tehniki izbire semen. Izbira semen je odvisna predvsem od domene, ki jo rešujemo. Ko so semena izbrana, sledi korak rasti regij. Razširimo regijo, ki ima sebi najbolj podobno sosednjo točko, ki še ni v regiji. Postopek ponavljamo tako dolgo, dokler ne zmanjka točk brez regije. Za funkcijo podobnosti se v večini primerov uporabita absolutna vrednost razlike povprečne intenzitete in intenziteta slikovne pike (3.2).

$$podobnost = |povprecna\_intenziteta\_regije - intenziteta\_slikovne\_pike| \tag{3.2}$$

Algoritem je bil zamišljen za segmentacijo slike, in ne za štetje objektov. Za štetje ni zanimiv, saj bi število semen določilo tudi končno število najdenih objektov. To za nas ne pride v poštev, saj semen ne znamo natančno določiti. Če bi jih, potemtakem rasti regij ne bi potrebovali. Ob tem tudi ni lahko izbrati semena, ki bi bilo ozadje. Pri gosto posejanih objektih tudi ni nujno, da je ozadje povezano. Zato smo iz algoritma pobrali samo za nas zanimiv koncept rasti regije, če ima regija zelo podobno sosedno točko.

Nastavitve ukaza (»ObjectCounterRegionGrowing«) opisuje tabela 3.5.

Tabela 3.5: Parametri ukaza

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
SeedThreshold	NE	150	Prag za določitev semen
MaxDiff	NE	10	Največja dovoljena razlika za dodajanje točke k regiji
MinSize	NE	0	Omejitev najmanjše velikosti
MaxSize	NE	100000	Omejitev največje velikosti
MinCirc	NE	0	Omejitev najmanjše kompaktnosti
MaxCirc	NE	1,0	Omejitev največje kompaktnosti
SizeRange	DA	/	Omejitev parametra MinSize in MaxSize

Števec za začetna semena upošteva točke, ki imajo večjo sivino od vrednosti nastavitve »SeedThreshold«. Spremeniti pa smo morali način rasti regij; regije razširjamo vsako posebej. Regiji točke dodajamo tako dolgo, kolikor ima sosednjo točko, ki ji je dovolj podobna. Dopustnost razlike za dodajanje točke k regiji določa nastavitev »MaxDiff«. Nastale regije ki se prekrivajo, združimo v eno skupino. Nato vse dobljene regije preštejemo enako, kot to dela osnovni števec delcev. Vse nastavitve v fazi optimizacije pretvorimo v parametre.

Zaradi časovne potratnosti algoritma smo morali naknadno dodati obvezno nastavitev »SizeRange«. Nastavitev deluje kot omejitev velikosti večanja regije. Če regija z rastjo preraste dovoljeno velikost, to regijo prenehamo povečevati in jo označimo kot neveljavno. S tem omejimo tudi najmanjšo oziroma največjo velikost parametra »MinSize« in »MaxSize« v fazi optimizacije. Pohitritev delovanja v fazi učenja je ogromna, saj se v fazi učenja pojavi veliko nesmiselnih kombinacij vrednosti nastavitev. Nesmiselne nastavitve lahko povzročijo stalno večanje vseh regij čez celotno sliko, kar je potratno opravilo.

Tabela 3.6: Parametri opcijskega ukaza.

Ime	Obvezen	Privzeta vrednost	Opis
ID	DA	/	Enolični identifikator
IsEnabled	NE	true	Ali se izvede blok stavkov

3.3 Kontrolni ukazi

V to sekcijo spadajo razširitve, ki niso nujno potrebne za štetje in optimizacijo. Z njimi lahko nadzorujemo potek izvajanja kode. Njihovo funkcionalnost bi lahko nadomestili z osnovnim jezikom ImageJ in parametri. Računalniškim nepoznavalcem omogočajo precej lažje nadzorovanje izvajanja kode in močno pripomorejo k berljivosti parametriziranega makroja ImageJ.

Implementirali smo ukaz, ki kodo izvede opcijsko (ukaz »Optional«). Parametri ukaza so opisani v tabeli 3.6. Če je parameter »IsEnabled« nastavljen na logično »1«, kodo v elementu izvedemo, drugače jo preskočimo. Nastavitev se v fazi optimizacije pretvori v parameter. Naloga genetskega algoritma bo ugotoviti, ali kodo izvesti ali ne.

3.4 Implementacija razširitev

Naš razširjeni jezik smo zasnovali tako, da ga je mogoče razmeroma preprosto razširiti z novimi gradniki. Vsaka razširitev implementira vmesnik Java »CodeElement« (vmesnik 3.1). Vmesnik predpisuje funkcije:

1. ***getTagName:*** vrne ime XML značke atributa;
2. ***getID:*** vrne enolični identifikator gradnika;
3. ***setID:*** nastavi enolični identifikator gradnika. Funkcija se izvaja pri razčlenjevanju razširjenega jezika ImageJ;
4. ***fillParameters:*** v seznam doda parametre, ki jih vsebuje gradnik. Funkcija se kliče pred optimizacijo, da pridobimo vse parametre, ki jih

vsebuje koda;

5. ***getExecutable:*** s to funkcijo ustvarimo izvršljivo kodo in vrniti mora izvršljivo kodo. Kot parameter dobi slovar, v katerem je shranjena ena konfiguracija izvajanja algoritma. Funkcija se kliče pred štetjem;

6. ***addCodeElementChild:*** v gradnik doda otroka. Če gradnik funkcionalno ne podpira možnosti dodajanja otrok, na tem mestu vrnemo napako. Funkcija se kliče v času razčlenjevanja;

7. ***getCodeElementChild:*** vrne otroka, ki je na mestu, ki ga določimo s parametrom;

8. ***getCodeElementChildCount:*** vrne število otrok gradnika;

9. ***setAttributes:*** dodatni posebni atributi, ki jih posredujemo gradniku.

Shema 3.1: Vmesnik "CodeElement"

```
1  public interface CodeElement {
2      public String getTagName();
3      public String getID();
4      public void setID(String id);
5      public void fillParameters(List<Parameter> parameterList);
6      public String getExecutable(Map<Parameter, String>
           parameterValues);
7      public void addCodeElementChild(CodeElement codeElement);
8      public CodeElement getCodeElementChild(int index);
9      public int getCodeElementChildCount();
10     public void setAttributes(Attributes attributes);
11 }
```

Za implementacijo parametrov je treba razširiti vmesnik »Parameter« (vmesnik 3.2). Vmesnik razširi »CodeElement« s funkcijami:

1. ***setValue:*** nastavi vrednost parametra;

2. ***getValue:*** vrne vrednost parametra;

3. ***valueFromString:*** pridobi vrednost iz znakovne predstavitve, potrebuje se v fazi razčlenjevanja;

4. ***stringFromValue:*** iz vrednosti ustvari niz, ki se potrebuje pri shranjevanju v datoteko;

5. ***getGeneImplementation:*** vrne implementacijo parametra, ki se uporabi v fazi optimizacije. Ker smo za ogrodje genetskega algoritma uporabili JGAP, je treba na tem mestu vrniti JGAP-implementacijo parametra [6];

6. ***getParent:*** vrne element, ki je ustvaril parameter. Če gre za samostojni parameter, vrne vrednost »null«;

7. ***setParent:*** funkcija nastavi element, ki je ustvaril parameter. Kliče jo gradnik na interno ustvarjenih parametrih.

Shema 3.2: Vmesnik "Parameter"

```
1 public interface Parameter extends CodeElement {
2     public void setValue(Object value);
3     public Object getValue();
4     public Object valueFromString(String value);
5     public String stringFromValue(Object value);
6     public Gene getGeneImplementation(Configuration conf);
7     public CodeElement getParent();
8     public void setParent(CodeElement codeElement);
9 }
```

Za implementacijo števcev je treba razširiti vmesnik »ObjectCounter« (vmesnik 3.3).). Vmesnik razširi »CodeElement« s funkcijami:

1. ***getPoints:*** funkcija vrne lokacije objektov, ki jih je preštel števec. Prvi parameter je slika, na kateri štejemo objekte, drugi parameter pa je slovar vrednosti parametrov.

Shema 3.3: Vmesnik "ObjectCounter"

```
1 public interface ObjectCounter extends CodeElement {
2     public ImagePoint[] getPoints(ImagePlus imagePlus, Map<Parameter,
          String> parameterValues);
3 }
```

Poglavje 4

Optimizacija - genetski algoritem

Naša naloga je bila čim bolje nastaviti parametrizirani algoritem. Algoritmi imajo večinoma veliko parametrov, zato je prostor vseh možnih stanj ogromen. Pregled celotnega prostora tako ni izvedljiv v realnem času. Čim boljše nastavitve bomo poskušali nastaviti z genetskim algoritmom. Genetski algoritem je primeren za reševanje težav, kadar je kot hevristika na voljo le presoja kakovosti posamezne rešitve. Na podlagi te hevristike pametneje preiskuje prostor stanj. Zagotovila, da bomo dobili najboljšo možno rešitev, ni, bomo pa z veliko manjšim preiskanim prostorom stanj verjetno dobili razmeroma dovolj dobro rešitev. Manjši preiskani prostor stanj pomeni krajši čas izvajanja optimizacije.

Zamisel in zasnova genetskega algoritma posnemata razvoj živih bitij. Navdihnjeni sta z evolucijo in naravnim izborom. Psevdokoda je prikazana v algoritmu 3. Na začetku naključno ustvarimo populacijo osebkov (1. vrstica). Posamezni osebek predstavlja eno možno rešitev našega problema. Nato pa, dokler ni izpolnjen ustavitveni pogoj, ponavljamo:

1. ***križanje osebkov:*** iz dveh možnih osebkov s križanjem ustvarimo novega;

2. ***mutacijo osebkov:*** del osebkov naključno spremenimo;

3. ***evalvacijo populacije:*** izračunamo kakovost osebkov v populaciji;

4. ***naravni izbor:*** šibke osebke zavržemo iz populacije.

Genetski algoritem ni natančen predpis. Vsebuje le smernice, kako implementirati algoritem preiskovanja prostora stanj. Zato se je razvilo veliko različic algoritma; te ponujajo splošne izboljšave oziroma izboljšave za preiskovanje domen z določenimi lastnostmi.

Algoritem 3: Psevdokoda genetskega algoritma [7]

```
Inicializacija in evalvacija začetne populacije
while ustavitveni pogoj ni izpolnjen do
    križanje osebkov
    mutacija osebkov
    evalvacija populacije
    naravni izbor
end while
```

Zelo dobra lastnost je ta, da je za pripravo genetskega algoritma za delovanje potrebnega razmeroma malo algoritmičnega in matematičnega znanja o problemu. Za delovanje je treba opredeliti:

- definicijo osebka,
- kriterijsko funkcijo,
- velikost populacije,
- ustavitveni pogoj,
- križanje osebkov,
- mutacijo osebkov,
- naravni izbor.

4.1 Definicija osebka

Posamezno možno rešitev problema predstavlja osebek. Osebek vsebuje množico nastavitev, imenovano kromosom. Kromosom vsebuje osnovne nastavitve, imenovane gen. V našem primeru bo videz kromosoma odvisen od algoritma števca. Vsak parameter v algoritmu pomeni en gen v kromosomu (4.1).

$$K = \{param_1,\ param_2,\ \cdots,\ param_i\} \tag{4.1}$$

Vrednosti genov (4.2) določijo vrednosti parametrov, ki se uporabijo v času izvajanja algoritma.

$$V = \{value_1,\ value_2,\ \cdots,\ value_i\} \tag{4.2}$$

4.2 Kriterijska funkcija

Da bi dobili želene rezultate, je treba dobro opredeliti hevristiko. Ta bo poskrbela, da se bo populacija osebkov razvijala v želeno smer. Zato je treba znati ovrednotiti kakovost osebkov. Osebke ovrednotimo s kriterijsko funkcijo. Ta funkcija mora v našem primeru opredeliti, kako dobro nastavitve osebka preštejejo objekte na množici slik.

Naša kriterijska funkcija kot parameter pričakuje množico (4.3) n-tih slik, ki jih bomo uporabili za učenje.

$$IMG = \{img_1,\ img_2,\ \cdots,\ img_n\} \tag{4.3}$$

Za vsako sliko moramo določiti lokacije ročno preštetih objektov (4.4). To množico bomo uporabili kot resnično vrednost (»ground truth«).

$$C_{man}(img) = \{p_1,\ p_2,\ \cdots,\ p_n\} \tag{4.4}$$

Z vrednostmi parametrov (4.2) lahko za vsako sliko pridobimo izračunane lokacije objektov (4.5).

$$C_{calc}(img,\ V) = \{p_1,\ p_2,\ \cdots,\ p_n\} \tag{4.5}$$

Skupno uspešnost osebka S bomo v našem primeru opredelili kot vsoto uspešnosti za posamezno sliko S_{img} (4.6).

$$S(V) = \sum_{i=1}^{n} S_{img_i}(V) \tag{4.6}$$

Opredeliti uspešnost števca ni preprosto. Zatakne se že pri opisu opredelitve. Veliko lažje je razmišljati o napaki, ki jo števec naredi. V nadaljevanju bomo zato poskušali čim bolje določiti napako števca E_{img}. Uspešnost osebka iz napake bomo dobili kot obratno vrednost (4.7).

$$S_{img}(V) = \frac{1}{E_{img}(V)} \tag{4.7}$$

Sledila je opredelitev napake števca za posamezno sliko. Da bi lahko izračunali napako števca, moramo za vsako sliko poznati množico ročno preštetih objektov (C_{man}) in množico programsko preštetih objektov (C_{calc}). Opredeliti napako števca je težavno. Zatakne se tudi človeku. Človek to dela z občutkom in poznavanjem problema, ki ga s števcem želi rešiti. Presojo bi izvedel s primerjanjem, in ne z vrednotenjem posameznega števca. Objektivno to spremeniti v formulo je dejansko nemogoče. Kakšen števec želimo, je precej odvisno tudi od semantike problema, ki ga naš števec želi rešiti. Obstajajo problemi, pri katerih smo lahko zelo občutljivi na lažno pozitivne objekte štetja. Kriterijsko funkcijo bomo poskusili čim bolj posplošiti, da bo dejansko splošno dobra za čim večjo domeno problemov štetja.

Naša prva predlagana napaka je bila absolutna napaka (4.8). Ta napaka je absolutna razlika med ročno preštetimi in izračunanimi točkami.

$$E_{img}(V) = \left| \mathrm{m}(C_{calc}(img,\ V)) - \mathrm{m}(C_{man}(img)) \right| \tag{4.8}$$

Funkcija absolutne napake deluje dobro pri večini domen. Matematikom je absolutna razlika za dokazovanje in izpeljave neželena. Namesto absolutne razlike raje uporabljajo kvadratno funkcijo; zanimiva jim je predvsem zaradi odvajanja. Pri štetju slik se kvadratna funkcija ni dobro obnesla, saj je izredno kaznovala števce, ki so na neki sliki naredili veliko napako. Šlo je za

iskanje števca z najmanjšo maksimalno napako. Slabost absolutne napake se je pokazala v primerih, ko se število iskanih objektov po posameznih slikah zelo razlikuje. V tem primeru smo zanemarili napako na slikah, ki vsebujejo malo objektov, saj se preprosto izgubijo v napakah pri slikah, ki vsebujejo veliko objektov. Zato smo se odločili za relativno napako (4.9). To napako smo dobili tako, da smo absolutno napako normirali glede na število ročno preštetih objektov.

$$E_{img}(V) = \left| \frac{\mathrm{m}(C_{calc}(img,\ V)) - \mathrm{m}(C_{man}(img))}{\max(1, \mathrm{m}(C_{man}(img)))} \right| \tag{4.9}$$

Čeprav smo informacijo o lokacijah objektov popolnoma zanemarili, se je tako opredeljena napaka dobro obnesla pri večini mikroskopskih slik. Te slike so razmeroma dobro predvidljive, saj so posnete v nadzorovanih okoljih. Težava je nastala pri domenah slik, ki niso iz domene mikroskopa in so zato veliko bolj raznolike. Tako se nam je zgodilo, da kriterijska funkcija ni zaznala, da je nekatere objekte štela večkrat. Pri zelo raznoliki domeni slik se nam je celo zgodilo, da števec niti približno ni našel lokacije pravih objektov in je le po naključju našel približno pravo število objektov. Zato smo morali v napako dodati še povprečno napako razdalje, ki take števce ustrezno kaznuje (4.10).

$$\begin{aligned} E_{img}(V) = &\left| \frac{\mathrm{m}(C_{calc}(img,\ V)) - \mathrm{m}(C_{man}(img))}{\max(1, \mathrm{m}(C_{man}(img)))} \right| \\ &+ DistanceError(C_{calc}(img,\ V),\ C_{man}(img)) \end{aligned} \tag{4.10}$$

Zato smo opredelili funkcijo »DistanceError« (algoritem 4). Funkcija na vhodu pričakuje dve množici točk, med katerima izračuna napako razdalje. Prva množica bodo v našem primeru ročno preštete točke, druga pa izračunane točke. Funkcija »NearestPoint« vrne točki, ki sta si med tema dvema množicama najbližje. Funkcija je požrešna in zato ni nujno, da dobimo najmanjšo možno vsoto napak razdalje. Algoritem smo poenostavili zaradi velikega prihranka računske zahtevnosti. Napako smo normalizirali z diagonalo slike; s tem smo lahko enakovredno sešteli povprečno napako raz-

dalje z relativno napako števila objektov. Poleg tega je normalizacija rešila problem različnih velikost slik.

Algoritem 4: Psevdokoda napake razdalje

```
1: function DistanceError(C_calc, C_man)
2:     pairCount ← min(m(C_calc), m(C_man))
3:     sumError ← 0
4:     while C_calc ≠ ∅ and C_man ≠ ∅ do
5:         point_{C_calc}, point_{C_man} = NearestPoint(C_calc, C_man)
6:         curError ← dist(point_{C_calc}, point_{C_man}) / image_diagonal
7:         sumError ← sumError + curError
8:         C_calc ← C_calc \ {point_{C_calc}}
9:         C_man ← C_man \ {point_{C_man}}
10:    end while
11:    return sumError / max(1, pairCount)
12: end function
```

Pri seštevanju pomensko različnih stvari, se srečamo s problemom obteževanja seštevancev. V našem primeru gre za obteževanje relativne napake in napake razdalje. Mi smo uteži pustili privzete. V praksi je to pomenilo, da smo večjo veljavo dali relativni napaki. Splošno dobro določiti uteži je težavno, saj je določitev odvisna tudi od domene. Problem bi lahko rešili tako, da bi relativno napako in napako razdalje izračunali in obravnavali ločeno.

4.3 Velikost populacije

Velikost populacije je število osebkov v populaciji. Večja velikost populacije pomeni večjo raznolikost osebkov v njej. Prevelika velikost populacije upočasni genetski algoritem. Premajhna pa poveča možnost, da obtičimo v lokalnem optimumu.

V našem primeru je računanje kakovosti osebka izredno zamudna operacija. Algoritmi za obdelavo slik so časovno zelo potratni. Ob tem je treba

algoritem pognati na vseh učnih slikah. Zato smo za genetske algoritme privzeto izbrali razmeroma majhno populacijo s sto osebki.

4.4 Ustavitveni pogoj

Z ustavitvenim pogojem določimo, kdaj smo z rešitvijo zadovoljni oziroma se je napredek iz generacije v generacijo ustavil. Prestrog ukinitveni kriterij povzroči prehitro končanje genetskega algoritma, kar prinese slabše končne rezultate. Premalo strog ustavitveni kriterij pa povzroči, da se genetski algoritem praktično nikoli ne bo končal. V praksi so najpogostejši ustavitveni pogoj med drugim:

- fiksno število generacij,
- fiksni čas izvajanja algoritma,
- dosežen prag uspešnosti,
- konvergenca uspešnosti najboljšega osebka.

Za naš problem optimizacije algoritma čas izvajanja optimizacije ni ključen. Pomembneje je, kako dobro rešitev dobimo. Tako je naš ustavitveni pogoj izpolnjen, ko deset generacij zaporedoma ne dobimo boljše rešitve. Seveda imamo možnost, da sami predčasno končamo izvajanje.

4.5 Naravni izbor

V naravi imajo močnejši osebki večjo verjetnost preživetja. Enako moramo to simulirati tudi pri genetskem algoritmu. To bomo dosegli tako, da bomo vsako generacijo slabe osebke zavrgli. S tem poskrbimo, da se splošna kakovost populacije iz generacije v generacijo izboljšuje. V našem primeru v vsaki generaciji obdržimo 90 % najboljših osebkov [6].

4.6 Križanje osebkov

V tej fazi simuliramo razmnoževanje osebkov, pri čemer se kombinirajo geni dveh osebkov. Obstaja veliko načinov križanja. Osnovna različica je, da naključno izberemo dva osebka, iz katerih bomo razvili dva nova. Naključno izberemo še mesto križanja osebka. Nova osebka tako dobimo s kombinacijama prvega in drugega dela izvornih osebkov. Nekatere implementacije lahko predpisujejo več točk križanja. Število križanj na posamezno generacijo (4.11) je v našem primeru odvisno od števila osebkov v populaciji [6].

$$stevilo\_krizanj = 35\% \cdot velikost\_populacije \tag{4.11}$$

4.7 Mutacija osebkov

Pri križanju DNK-a se pojavljajo napake (mutacije). Podobno funkcionalnost mora uporabljati tudi genetski algoritem. Samo križanje osebkov je premalo. S križanjem preizkušamo le različne kombinacije, nič pa ne naredimo za to, da bi preiskali večji prostor stanj. Mutacija se izvaja tako, da osebkom v celotni populaciji naključno spremenimo katerega izmed genov. V našem primeru na vsakem 12. genu izvedemo mutacijo. Število mutacij (4.12) je v našem primeru odvisno od velikosti populacije in število genov v kromosomu [6]:

$$stevilo\_mutacij = \frac{1}{12} \cdot velikost\_populacije \cdot velikost\_kromosoma \tag{4.12}$$

Poglavje 5

Aplikacija za iskanje števcev objektov

Naš želeni rezultat je uporabna aplikacija. Ta mora omogočati:

- optimizacijo parametrov algoritma,
- poganjanje optimiziranega števca na slikah.

Aplikacijo lahko poganjamo prek uporabniškega vmesnika ali ukazne vrstice. Oba načina poganjanja za svoje delovanje potrebujeta projektno datoteko. Ta datoteka je namenjena hrambi podatkov aplikacije in vsebuje podatke za eno domeno slik. Format datoteke je XML. Vsebina datoteke je opisana v shemi 5.1. S shemo želimo projektno datoteko narediti dostopno zunanjim aplikacijam. Tako lahko zunanje aplikacije same sestavijo projektno datoteko ali iz nje pridobivajo rezultate štetja. Koren XML datoteke je element »Project«. Koren vsebuje elemente:

1. ***Algorithm:*** element vsebuje števec, ki je spisan v parametriziranem makro jeziku ImageJ;
2. ***Image:*** vsebuje slike, na katerih želimo poganjati algoritem. Ko algoritem poženemo, se v okviru slik napolni seznam izračunanih točk (»CalcPoint«);

3. ***LearnImage:*** vsebuje slike, ki so namenjene optimizaciji parametrov. Slike morajo imeti napolnjen seznam ročno preštetih objektov (»RealPoint«). S parametrom »LearnImage« pa določimo, ali bo slika v fazi optimizacije namenjena učenju ali preizkušanju števca. Izračunane točke se zapišejo v element »CalcPoint« in se pozneje uporabijo za izračun napake testne ali učne množice slik.

Shema 5.1: Shema XSD projekta

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Project">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Algorithm" type="xs:string" />
        <xs:element name="Image" type="Image" minOccurs="0"
            maxOccurs="unbounded" />
        <xs:element name="LearnImage" type="LearnImage"
            minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="Image">
    <xs:sequence>
      <xs:element name="Path" type="xs:string" />
      <xs:element name="CalcPoint" type="ImagePoint"
          minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="LearnImage">
    <xs:sequence>
      <xs:element name="Path" type="xs:string" />
      <xs:element name="IsLearnImage" type="xs:boolean" />
      <xs:element name="RealPoint" type="ImagePoint"
          minOccurs="0" maxOccurs="unbounded" />
```

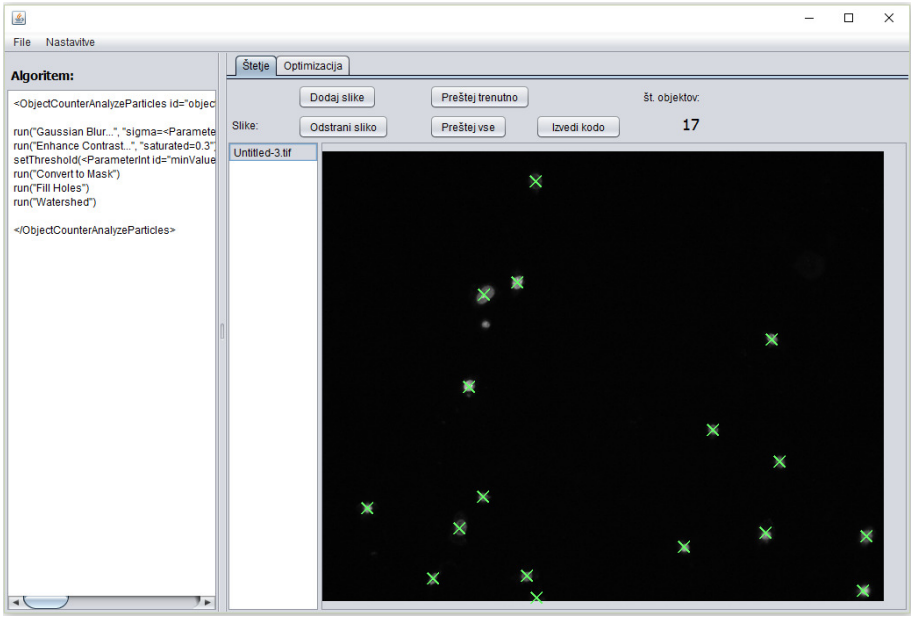
```
25          <xs:element name="CalcPoint" type="ImagePoint"
                minOccurs="0" maxOccurs="unbounded" />
26        </xs:sequence>
27      </xs:complexType>
28
29      <xs:complexType name="ImagePoint">
30        <xs:sequence>
31          <xs:element name="X" type="xs:int" />
32          <xs:element name="Y" type="xs:int" />
33        </xs:sequence>
34      </xs:complexType>
35  </xs:schema>
```

5.1 Uporabniški vmesnik

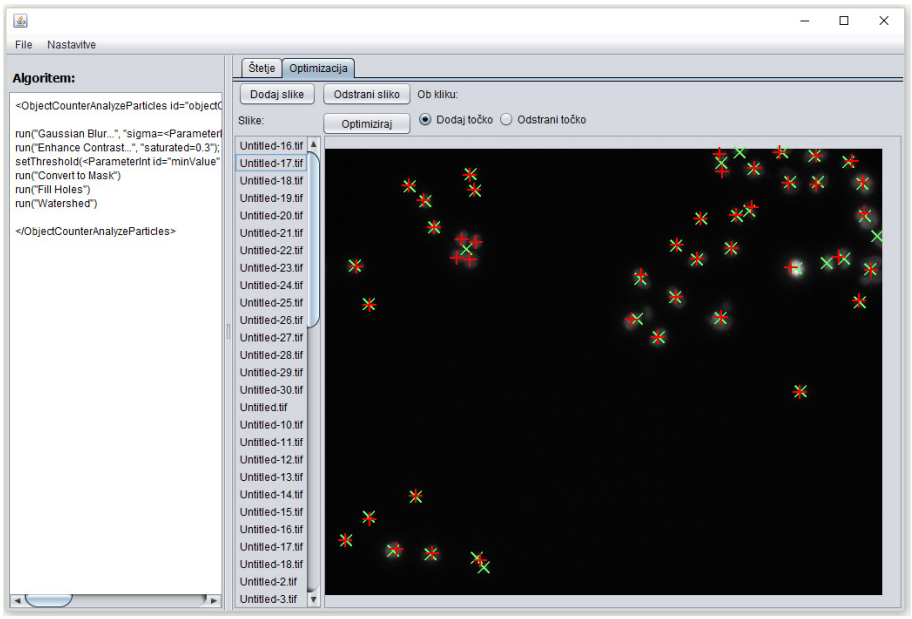
Razvili smo uporabniški vmesnik, ki omogoča lažjo uporabo optimizacijskega algoritma. Na levi strani uporabniškega vmesnika je na voljo vnosno polje. V to vnosno polje vnesemo parametrizirani genetski algoritem, ki bo štel izbrano domeno. Na desni strani sta na voljo zavihka »štetje« in »optimizacija«. Zavihek štetje je namenjen poganjanju algoritma (slika 5.1). Pred poganjanjem moramo najprej dodati slike, na katerih bomo izvajali štetje objektov. Slike dodamo z gumbom »Dodaj sliko«, ki odpre dialog za izbiro slik na datotečnem sistemu. Štetje poženemo z ukazoma »Poženi na trenutni« ali »Poženi na vseh«. Za vsako prešteto sliko prikažemo število najdenih objektov in te objekte tudi označimo na sliki. Uporabo smo razširili z ukazom »Poženi algoritem«, ki se obnaša enako kot ImageJ. S tem smo zavihek naredili uporaben tudi v fazi razvoja algoritma.

Zavihek »Optimizacija« (slika 5.2) je namenjen optimizaciji števca z genetskim algoritmom. V tem zavihku najprej dodamo slike, ki jih želimo uporabili za učenje algoritma. Na vseh izbranih slikah moramo pred optimizacijo označiti vse iskane objekte. Objekte označimo s klikanjem po sliki. Množica slik za učenje naj bo za dobre in realne rezultate čim bolj raznolika.

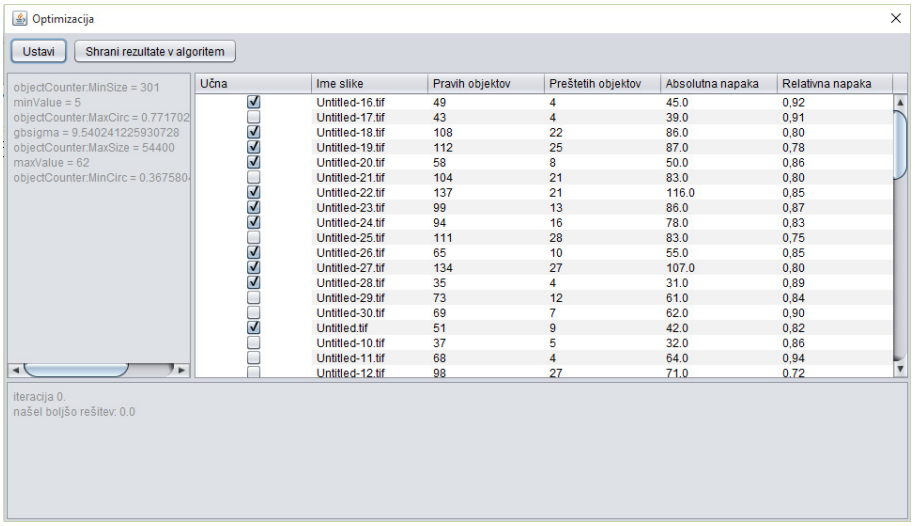
Ko označimo vse objekte na sliki in smo zadovoljni z algoritmom, pri-



Slika 5.1: Zaslonska slika zavihka »Štetje«



Slika 5.2: Zaslonska slika zavihka »Optimizacija«



Slika 5.3: Zaslonska slika dialoga optimizacije

tisnemo gumb »Optimiziraj«. Odpre se dialog, v katerem se bodo v času optimizacije izpisovali vmesni rezultati (slika 5.3). Ko se dialog odpre, se v tabeli izpišejo vse slike, ki smo jih ročno označili. V prvem stolpcu je potrditveno polje, s katerim izberemo, ali bo slika namenjena učenju ali preizkušanju. Privzete vrednosti za posamezno sliko program izbere naključno. 60 % slik namenimo učenju, 40 % pa preizkušanju. Če želimo ugotoviti realno uspešnost števca, moramo preizkušanje izvesti na množici slik, ki ni bila v fazi učenja nikoli uporabljena. Učna množica za rezultate ni relevantna, saj smo se v fazi učenja njej prilagajali. Možnost, kateri množici bo slika pripadala, lahko pred izvajanjem še spremenimo. Na levi strani se izpisujejo vrednost parametrov do zdaj najboljše konfiguracije. V polju spodaj pa se izpisujejo dodatne informacije glede izvajanja. Če smo z rezultatom zadovoljni oziroma želimo prekiniti izvajanje, je na voljo gumb »Ustavi«, s katerim se ustavi izvajanje. Gumb »Shrani rezultate v algoritem« prenese vrednosti parametrov trenutno najboljše konfiguracije v algoritem.

5.2 Ukazna vrstica

Za naprednejše uporabnike smo podprli možnost poganjanja prek ukazne vrstice. S tem želimo podobno kot ImageJ še dodatno razširiti možnosti uporabe. Trenutno podprta ukaza sta »run« in »optimize«.

Ukaz »run« za vse slike v projektu prešteje število objektov. Ukaz kot parameter pričakuje pot do projektne datoteke. V projektu morajo biti opredeljeni algoritem in slike, na katerih želimo prešteti število objektov. Rezultat se shrani v seznam izračunanih točk (»CalcPoint«).

Ukaz »optimize« požene optimizacijo parametrov. Kot prvi parameter pričakuje pot do projektne datoteke. Ukaz v projektu pričakuje opredeljen algoritem in seznam ročno preštetih učnih slik. Za vsako učno sliko moramo določiti še, ali jo bomo uporabili za učenje ali preizkušanje. Rezultat operacije je algoritem, nastavljen z optimalnimi vrednostmi parametrov.

Poglavje 6

Testiranje

Testiranje smo izvajali v našem uporabniškem vmesniku. Za vsako domeno slik smo naredili nov projekt in v ta projekt smo dodali učne slike. Za vse učne slike smo ročno označili objekte, ki jih želimo prešteti. Označeni objekti pomenijo resnično vrednost (»ground truth«). Težava našega testiranja je, da števcev za naše testne domene ni. Zato naše rešitve ni mogoče primerjati z morebitnimi drugimi.

Sledil je najzahtevnejši in najpomembnejši korak: pisanje parametriziranega makro števca ImageJ. Od kakovosti algoritma je močno odvisna končna kakovost rešitve. Ko smo bili s števcem zadovoljni, smo pognali optimizacijo. Optimizacijo smo pognali na naključnih 60 % ročno preštetih slik (učna množica).

Po končani optimizaciji je sledila analiza na novo dobljenega števca. Za analizo števca smo uporabili preostalih 40 % ročno preštetih slik (testna množica). Teh slik v fazi učenja nismo nikoli uporabili. Neuporaba testne množice zagotavlja verodostojnost analize kakovosti števca.

Nato smo z novim števcem na celotni testni množici pognali štetje. Za vsako sliko smo izračunali absolutno in relativno napako štetja. Vrednosti relativnih napak tvorijo približno normalno porazdelitev. Da bi natančno določili normalno porazdelitev, moramo poznati povprečno vrednost relativne napake in njen standardni odklon. Normalna porazdelitev pove, s ko-

likšno verjetnostjo se v nekem intervalu pojavlja relativna napaka štetja. Za nas je bil bolj zanimiv kumulativni diagram normalne porazdelitve. Iz tega diagrama bo razvidno, s kolikšno verjetnostjo je naša relativna napaka manjša ali enaka neki vrednosti.

6.1 Opis domen

Za preizkušanje genetskega algoritma smo potrebovali čim bolj raznolike domene slik. Želja je bila, da bi bili to realni problemi, za katere se dejansko potrebuje števec. Naša prva domena so bile slike celic kitajskega hrčka, pri katerih so s štetjem želeli določiti delež mrtvih celic. Za drugo domeno smo iz Kliničnega centra dobili mikroskopske celice fibroblastov. S štetjem so želeli določiti uspešnost razmnoževanja celic pod različnimi zunanjimi vplivi. Tretjo domeno smo dobili iz Biotehniške fakultete. V mikroskopski sliki spodnjega površinskega tkiva teloha so želeli prešteti reže. S številom rež lahko določijo pogoje za rast. Za zadnjo domeno smo želeli kakšno nemikroskopsko domeno. Izbrali smo slike jat ptic, na teh slikah smo želeli prešteti število ptic v jati.

Štetje na teh domenah so do zdaj izvajali ročno. Mi smo jim želeli to avtomatizirati, saj bi jim s tem prihranili veliko zamudnega in mučnega ročnega štetja objektov. S testnimi domenami smo lahko odkrili pomanjkljivosti našega genetskega algoritma. Zaradi tega sta razvijanje in preizkušanje večinoma potekala vzporedno.

6.2 Določanje deleža mrtvih celic

Dobili smo dve množici slik, v katerih so bile slike celic kitajskega hrčka v celični kulturi. V prvi množici so bile slike obarvane s fluorescenčnim barvilom Hoechst 33342, ki obarva celična jedra (slika 6.1). V drugi množici pa so bile slike obarvane s propidijevim jodidom. Propidijev jodid obarva samo mrtve celice (slika 6.2). S kombinacijo obeh slik lahko določimo preživetje ce-

lic po tretiranju z naraščajočo koncentracijo nanodelcev [8]. Vsako množico slik smo obravnavali neodvisno v svojem projektu, saj gre za ločeno štetje.

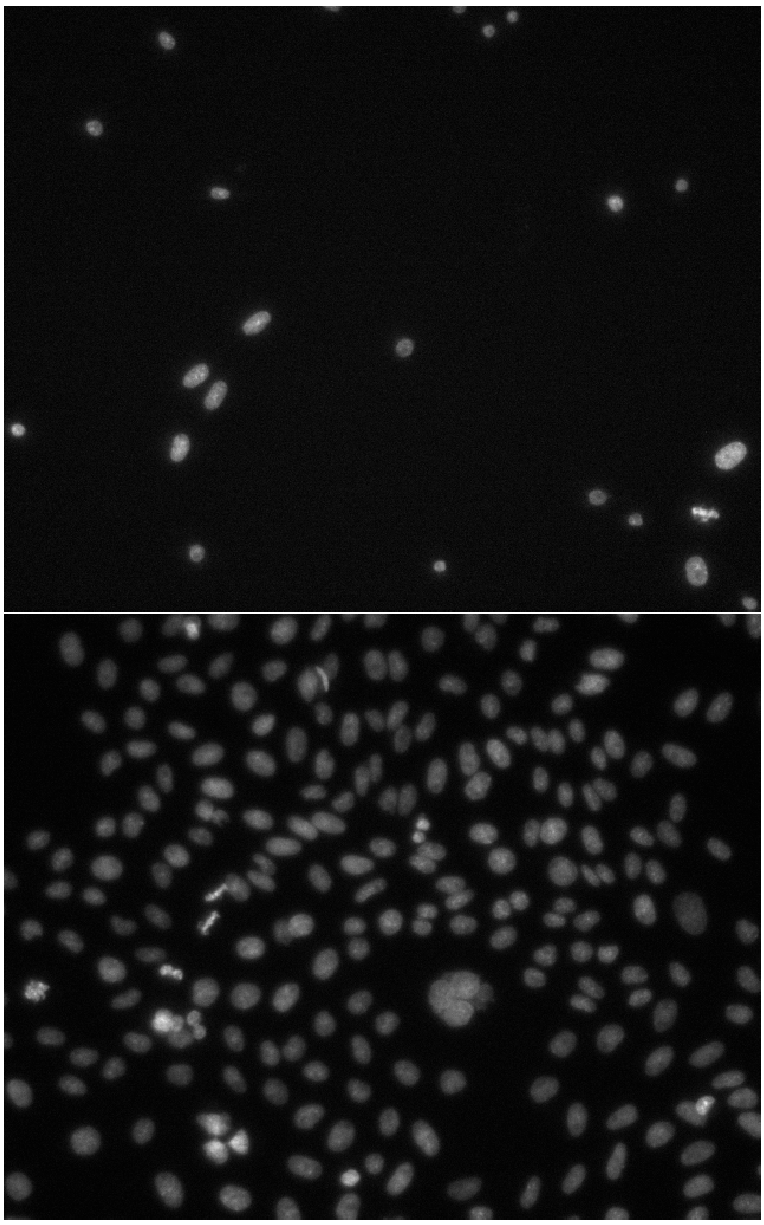
6.2.1 Algoritem

Slike v obeh množicah so sivinske. To pomeni, da vsebujejo en kanal intenzitete. Množici slik sta si med seboj zelo podobni. Zato bomo za obe uporabili enak algoritem (algoritem 5). Optimizacija pa bo poskrbela, da se bodo parametri prilagodili posamezni množici. Veliko različnih množic slik lahko preštejemo na zelo podoben način. Med seboj se razlikujejo le po nastavitvah parametrov. S parametrizacijo takih algoritmov omogočimo, da z izredno malo dela lahko podpremo novo domeno slik za štetje.

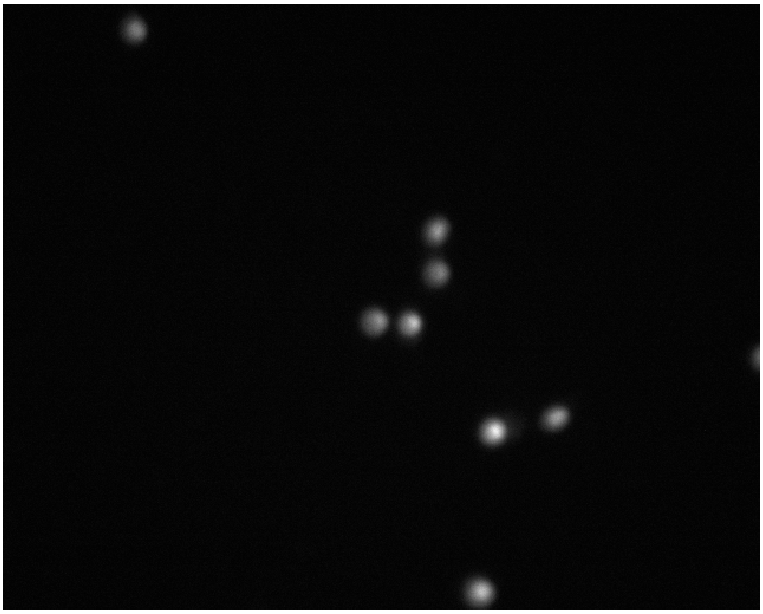
Algoritem 5: Psevdokoda štetja celic

```
1:  <ObjectCounterAnalyzeParticles id="objectCounter">
2:      run("Gaussian Blur...",
          "sigma=<ParameterDouble id="gbsigma"
          minValue="0.0" maxValue="10.0" />");
3:
4:      run("Enhance Contrast...", "saturated=0.3");
5:      setThreshold(
          <ParameterInt id="minTH" minValue="0" maxValue="255">,
          <ParameterInt id="maxTH" minValue="0" maxValue="255">)
6:
7:     run("Convert to Mask")
8:     run("Fill Holes")
9:     run("Watershed")
10: </ObjectCounterAnalyzeParticles>
```

Standardno je, da pred obdelavo slik te najprej zgladimo. Z glajenjem odstranimo morebiten šum s slik, ki lahko pokvari analizo. Glajenje lahko odstrani tudi nepomembne podrobnosti slike in tako omogoči lažjo nadaljnjo obdelavo slike. Za glajenje smo uporabili glajenje z normalno porazdelitvijo



Slika 6.1: Sliki celic, obarvanih z barvilom Hoechst 33342



Slika 6.2: Slika celic, obarvanih s propidijevim jodidom

(2. vrstica algoritma, funkcija »Gaussian Blur...«). Glajenje je izvedeno kot konvolucijski filter, katerega jedro je izračunano z normalno porazdelitvijo. Funkciji kot parameter podamo vrednost »sigma«, s čimer določimo standardni odklon normalne porazdelitve. Jedro je 2,5-krat večje od »sigme« in je normalizirano, da z operacijo ne vplivamo na povprečno svetlost slike. Vrednost »sigma« bo v našem primeru nastavil genetski algoritem, zato smo namesto vrednosti vstavili parameter.

Pri obdelavi slik izredno prav pride povečava kontrasta slike. Na nekaterih domenah slik že samo s povečavo kontrasta lahko zelo dobro izstopijo objekti, ki jih štejemo. S tem tudi izenačimo svetlost slik, kar naredi nadaljnjo obdelavo na množici slik preprostejšo. Za povečavo kontrasta smo uporabili vgrajeno funkcijo ImageJ »Enhance Contrast...« (4. vrstica algoritma). Funkcija raztegne histogram slike od najmanjše do največje možne vrednosti intenzitete točke. S parametrom »saturated« pa določimo, kolikšen delež točk lahko stisnemo v najmanjši ali največji intenziteti. S tem v izvorni sliki prezremo osamelce, ki izstopajo po svoji najmanjši ali največji

intenziteti. Ti osamelci bi preprečili zadostni razteg histograma.

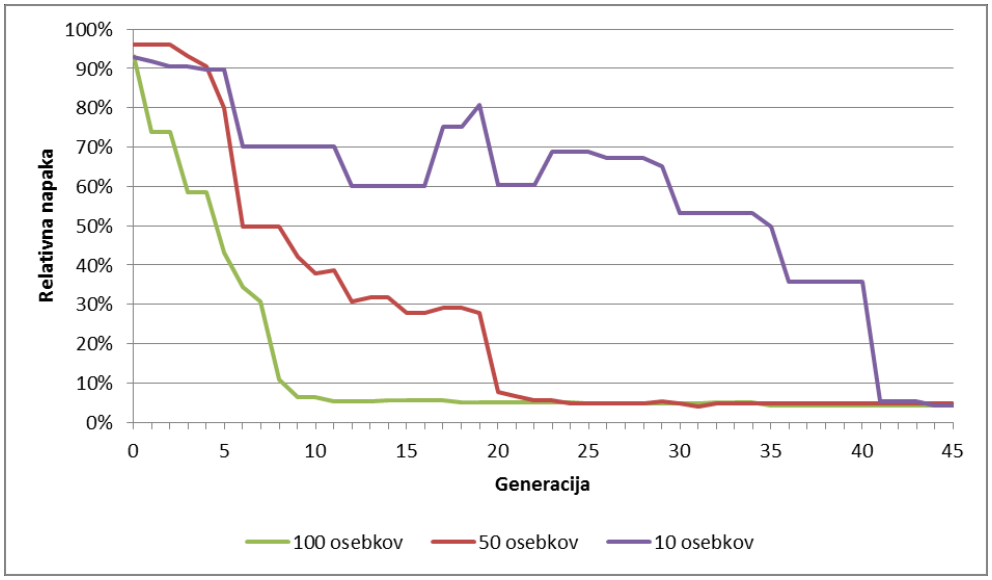
V naslednjem koraku sledi binarizacija slike. Binarizacijo v orodju ImageJ naredimo v dveh korakih. Najprej moramo nastaviti prag intenzitete (5. vrstica algoritma). V okviru praga bo slika imela logično »1«, zunaj pa logično »0«. Z ukazom »Convert to Mask« (7. vrstica algoritma) pa poženemo binarizacijo. Določanje optimalnega praga za celotno množico slik ni preprosto. Zato smo si na tem mestu pomagali s parametrizacijo in optimalne vrednosti je nastavil genetski algoritem.

Z ukazom »Fill Holes« (8. vrstica algoritma) zapolnimo luknje, ki so morebiti nastale v maski. Celice, ki se držijo skupaj, pa razdružimo z ukazom »Watershed« (9. vrstica algoritma). Tako obdelana slika je pripravljena za naš števec.

6.2.2 Optimizacija

Na množici slik, obarvanih z barvilom Hoechst 33342, smo opravili analizo napredka optimizacije. V analizi smo po generacijah spremljaji napredek relativne napake. Graf je prikazan na sliki 6.3. Prikazuje napredek za populacije velikosti 10, 50 in 100 osebkov.

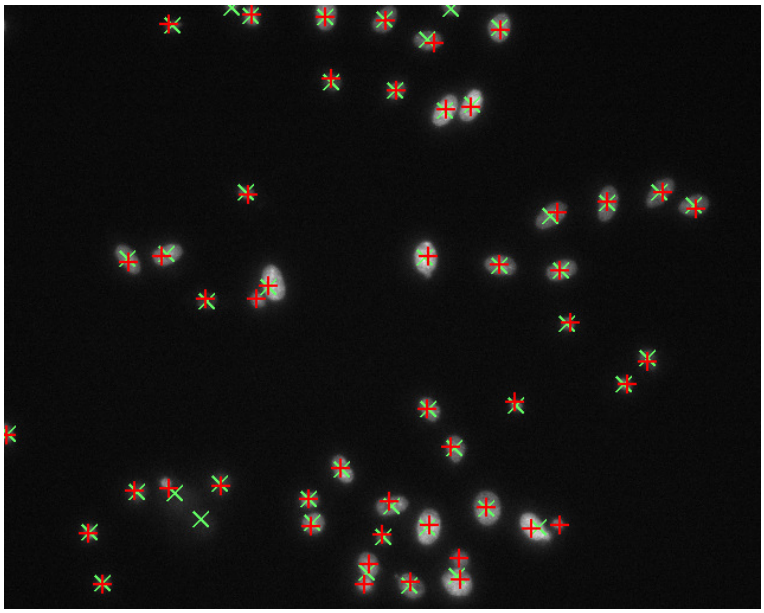
Pri vseh smo na koncu dobili podobne rezultate. Napredek se na neki stopnji ustavi, zato nadaljnje poganjanje optimizacije nima smisla. Ker gre za problem s sorazmerno malo atributi, smo po vsej verjetnosti pri vseh dosegli kar globalni optimum. Boljšega rezultata s trenutnim algoritmom ni več mogoče dobiti. Nič ni presenetljivo, če graf napake z novo generacijo tudi naraste. Ne velja, da boljša rešitev na učni množici prinese tudi boljšo rešitev na testni množici. Kot se iz grafa lepo vidi, za večje populacije potrebujemo manj iteracij, da najdemo boljšo rešitev. To je bilo tudi za pričakovati. Presenetil nas je razmeroma zelo dober napredek pri optimizaciji z 10 osebki. Za dober rezultat, smo skupno potrebovali veliko manj računanj uspešnosti osebkov. K temu verjetno pripomore večje število povratnih informacij. Smo pa mnenja, da bi se slabše odrezal pri težjih primerih z več atributi. Pri njih bi lahko obtičal v lokalnem optimumu.



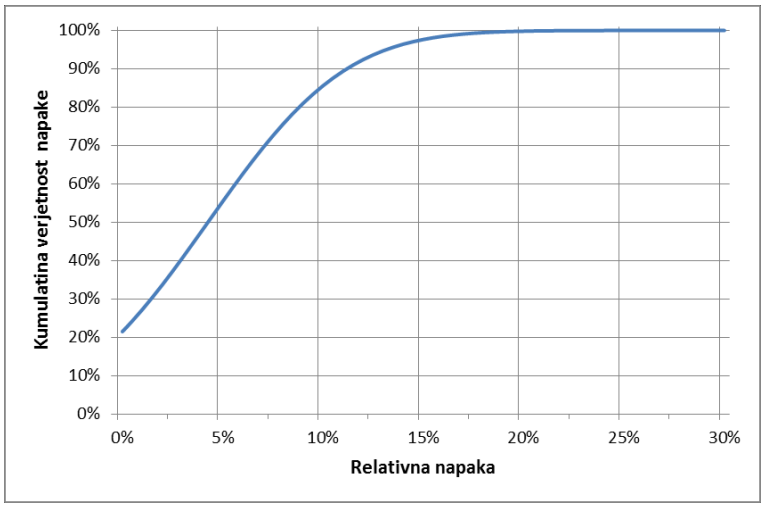
Slika 6.3: Grafični prikaz relativne napake po generacijah

6.2.3 Rezultati

Primer preštetih celic, obarvanih z barvilom Hoechst 33342, je prikazan na sliki 6.4. Z rdečim križcem so označene ročno preštete celice, z zelenim pa celice, preštete z algoritmom. Vrednost optimalnih parametrov je prikazana v tabeli 6.2. Rezultati na testni množici so prikazani v tabeli 6.1. Kot je razvidno iz tabele, so dobljeni rezultati izredno dobri. Naša povprečna relativna napaka je 4,26 %, s standardnim odklonom 5,4 %. Na podlagi kumulativnega diagrama normalne porazdelitve (slika 6.5) lahko ugotovimo, da imamo kar 85-odstotno možnost, da na sliki naredimo relativno napako, manjšo od 10 %.



Slika 6.4: Slika rezultatov števca celic, obarvanih z barvilom Hoechst 33342



Slika 6.5: Kumulativni diagram števca celic, obarvanih z barvilom Hoechst 33342

Tabela 6.1: Rezultati na testni množici (barvilo Hoechst 33342)

#	Število objektov	Število preštetih objektov	Absolutna napaka	Relativna napaka (v %)
1	49	48	1	0,02
2	43	43	0	0
3	112	108	4	0,04
4	134	133	1	0,01
5	69	72	3	0,04
6	91	93	2	0,02
7	24	22	2	0,08
8	64	62	2	0,03
9	35	36	1	0,03
10	63	62	1	0,02
11	117	120	3	0,03
12	137	142	5	0,04
13	106	112	6	0,06
14	156	155	1	0,01
15	104	107	3	0,03
16	155	157	2	0,01
17	171	194	23	0,13
18	121	124	3	0,02
19	153	160	7	0,05
20	166	170	4	0,02
21	98	97	1	0,01
22	80	82	2	0,03
23	30	32	2	0,07
24	207	215	8	0,04
25	181	187	6	0,03
26	229	229	0	0
27	334	240	94	0,28

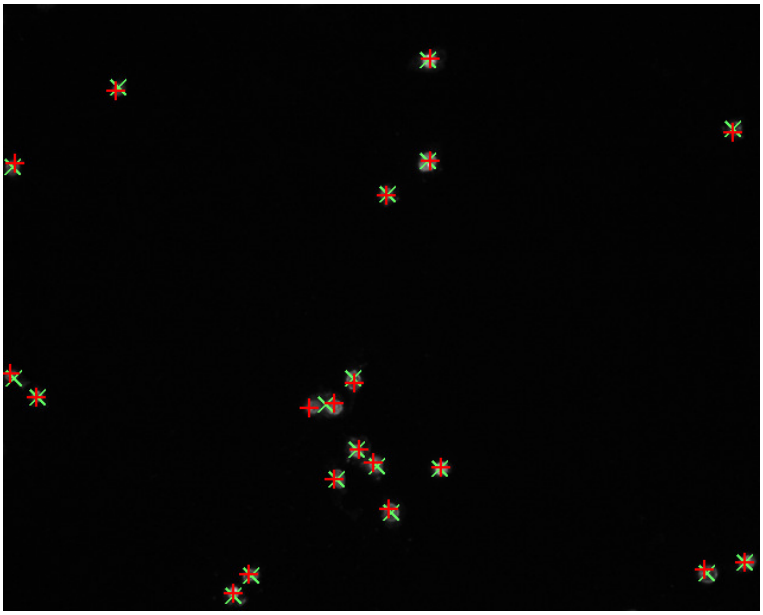
Tabela 6.2: Vrednost parametrov (barvilo Hoechst 33342)

Ime parametra	Vrednost parametra
objectCounter:MinSize	75
objectCounter:MaxSize	6508
objectCounter:MinCirc	$\approx 0,28$
objectCounter:MaxCirc	$\approx 0,99$
gbsigma	$\approx 1,84$
minTH	86
maxTH	253

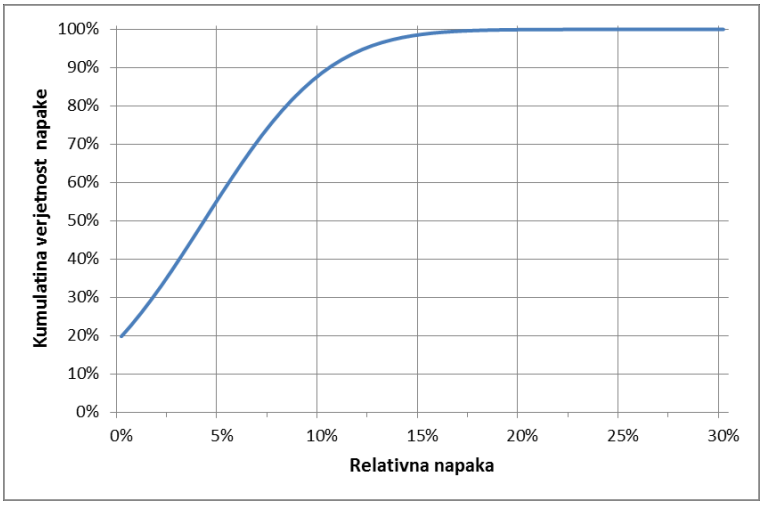
Primer preštetih celic, obarvanih s propidijevim jodidom, je prikazan na sliki 6.6. Z rdečim križcem so označene ročno preštete celice, z zelenim pa celice, preštete z algoritmom. Vrednost optimalnih parametrov je prikazana v tabeli 6.4. Rezultati na testni množici so prikazani v tabeli 6.3. Tudi ti dobljeni rezultati so dobri. Naša povprečna relativna napaka je 4,12 %, s standardnim odklonom 4,86 %. Imamo 93-odstotno možnost, da na sliki naredimo relativno napako, manjšo od 10 % (slika 6.7).

Pri tej skupini slik se je pokazala slabost relativne napake. Ta se je pokazala na rezultatih učne množice. V tej množici imamo večinoma slike z malo objekti (tudi slike brez objektov). Na učni množici na eni sliki nismo našli nobenega objekta, morali pa bi enega. Tako smo dobili 100-odstotno napako, kar je nekoliko pokvarilo skupni rezultat učne množice. Tako smo imeli povprečno napako učne množice 7,3 %, s standardnim odklonom 18,98 %.

Uporaba našega števca za določanje deleža mrtvih celic je bila odvisna od uspešnosti štetja v obeh množicah. Rezultati pri obeh množicah so bili dobri. Tako se je naš pristop izkazal primeren za določanje deleža mrtvih celic.



Slika 6.6: Slika rezultatov števca celic, obarvanih s propidijevim jodidom



Slika 6.7: Kumulativni diagram števca celic, obarvanih z barvilom Hoechst 33342

Tabela 6.3: Rezultati na testni množici (barvilo propidijev jodid)

#	Število objektov	Število preštetih objektov	Absolutna napaka	Relativna napaka (v %)
1	42	39	3	0,07
2	20	19	1	0,05
3	36	35	1	0,03
4	34	40	6	0,18
5	25	24	1	0,04
6	28	27	1	0,04
7	45	46	1	0,02
8	21	22	1	0,05
9	35	34	1	0,03
10	40	37	3	0,08
11	31	29	2	0,06
12	30	26	4	0,13
13	1	1	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0

Tabela 6.4: Vrednost parametrov (barvilo propidijev jodid)

Ime parametra	Vrednost parametra
objectCounter:MinSize	137
objectCounter:MaxSize	23109
objectCounter:MinCirc	$\approx 0,25$
objectCounter:MaxCirc	$\approx 0,95$
gbsigma	$\approx 0,61$
minTH	73
maxTH	231

6.3 Merjenje uspešnosti razmnoževanja fibroblastov

V Kliničnem centru, so z raziskavo v različnih pogojih želeli analizirati uspešnost razmnoževanja fibroblastov. Fibroblasti so celice (slika 6.8) v veznih tkivih. Veznemu tkivu dajejo moč, obliko in možnost povezovanja z drugimi tkivi; to jim omogoča njihova oblika.

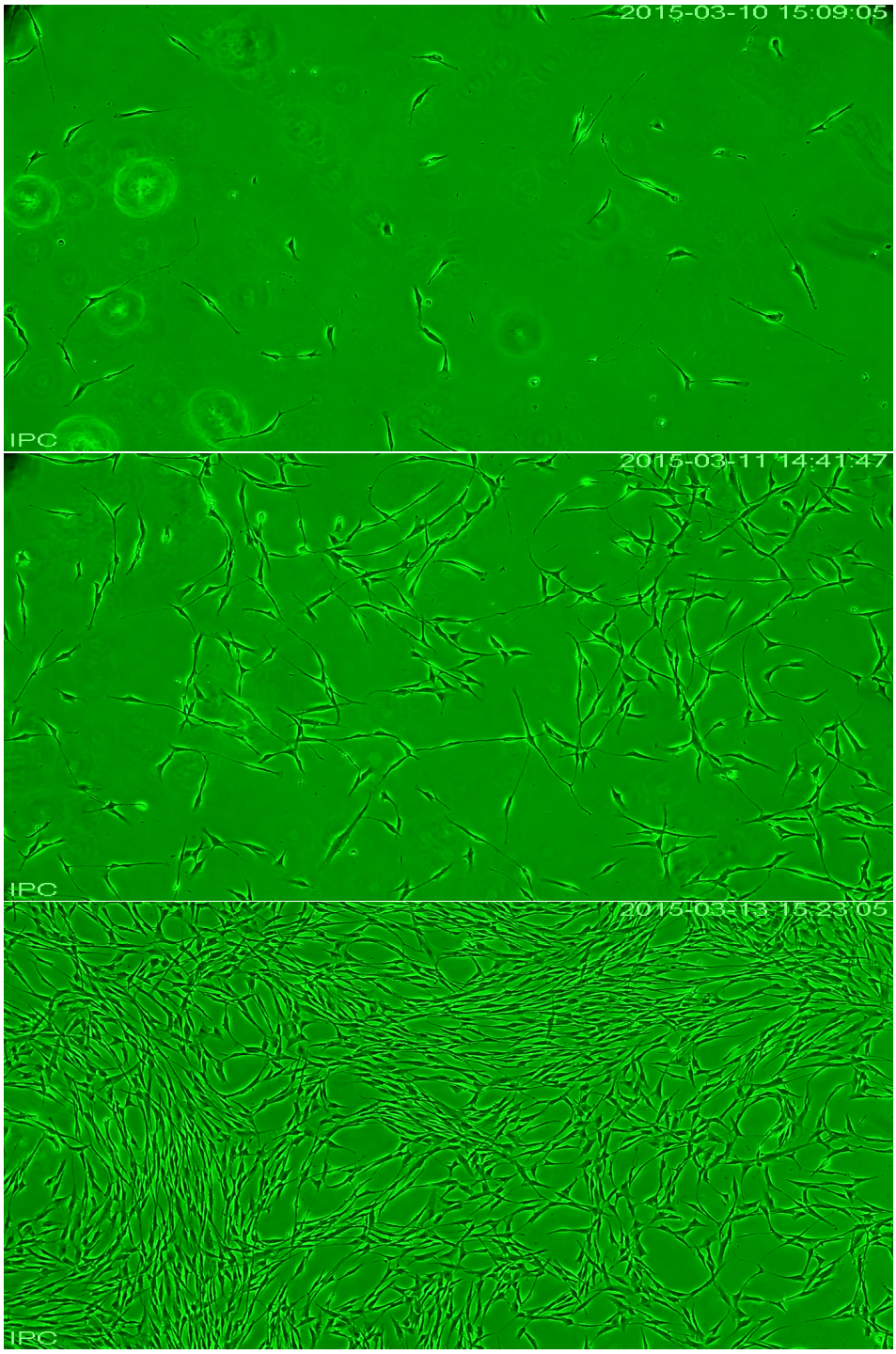
Razmnoževanje so pospeševali z obsevanjem nizkoenergetskega diodnega laserja Fotona XD-2 [9]. Uspešnost razmnoževanja so želeli proučiti pri obsevanju z različnimi močmi in pri različnih gostotah energije laserja. Zanimalo jih je število fibroblastov pred obsevanjem ter 24, 48 in 72 ur po obsevanju. Kot je razvidno na slikah, je štetje fibroblastov zahtevno. Celic je lahko veliko, saj lahko dosežejo izredno veliko gostoto. Na tej množici slik tudi ročno štetje pomeni veliko težavo. Zatakne se pri slikah, kjer so se fibroblasti izredno namnožili. Zato bo ta uspešnost štetja pomenila veliko olajšanje znanstvenikom, ki se s tem ukvarjajo.

6.3.1 Algoritem

Za štetje fibroblastov smo razvili parametrizirano kodo, ki jo prikazuje algoritem 6. Kot je standardno, smo sliko najprej zgladili z Gaussovim filtrom (3. vrstica algoritma). »Sigmo« normalne porazdelitve smo parametrizirali.

Slika je tipa RGB, kar pomeni, da vsebuje tri kanale. Večina algoritmov za obdelavo slik zna delati le na enem kanalu. Če jih že poženemo na več kanalih, vsak kanal obravnavajo ločeno. Eden takih je na primer glajenje z Gaussovim filtrom. Zaradi lažje nadaljnje obdelave in za pohitritev izvajanja je smiselno čim prej kanale združiti v enega. Za izračunani kanal si želimo, da čim bolj izpostavi iskane objekte. Za izračun lahko upoštevamo:

- povprečje vseh kanalov,
- svetlost slikovne pike, prilagojeno človekovi vidni zmožnosti; izračuna se kot $L = 0,3 \cdot R + 0,59 \cdot G + 0,11 \cdot B$,



Slika 6.8: Primeri slik

- različne kombinacije: na voljo so poljubne linearne kombinacije kanalov. Za potencialno dobre možne kombinacije lahko poskusimo tudi pretvorbo v različne barvne prostore (HSV, HSL, YPbPr itd).

V našem primeru so se objekti najbolje obnesli, ko smo s slike vzeli samo zeleni kanal. Postopek je prikazan v algoritmu v korakih od 5 do 16. Sledil je postopek večanja kontrasta z ukazom »Enhance Contrast...« (18. vrstica). Ukazu smo dodali še atribut »equalize«, ki izenači histogram za vse intenzitete.

Nato smo nastalo sliko poslali števcu z rastjo regij. Osnovni števec delcev je na tej množici slik popolnoma odpovedal. Osnovni števec na vhodu pričakuje binarno sliko. Ta binarizacija pa je na tej množici delala težave. Nikakor nam ni uspelo določiti samo celice. Ali smo označili preveč (slika 6.9, levo spodaj) ali smo označili premalo (slika 6.9, desno spodaj). Dobro pa se je videlo, da v primerih, ko ne označimo celotne celice, vseeno dobro označimo njene dele. Težava je bila samo v določitvi, kaj je zdaj ena celica. Zato smo uporabili števec z rastjo regij. Pri tem upamo, da je rast posameznih delov celice (semena) pravilno določila celotno celico.

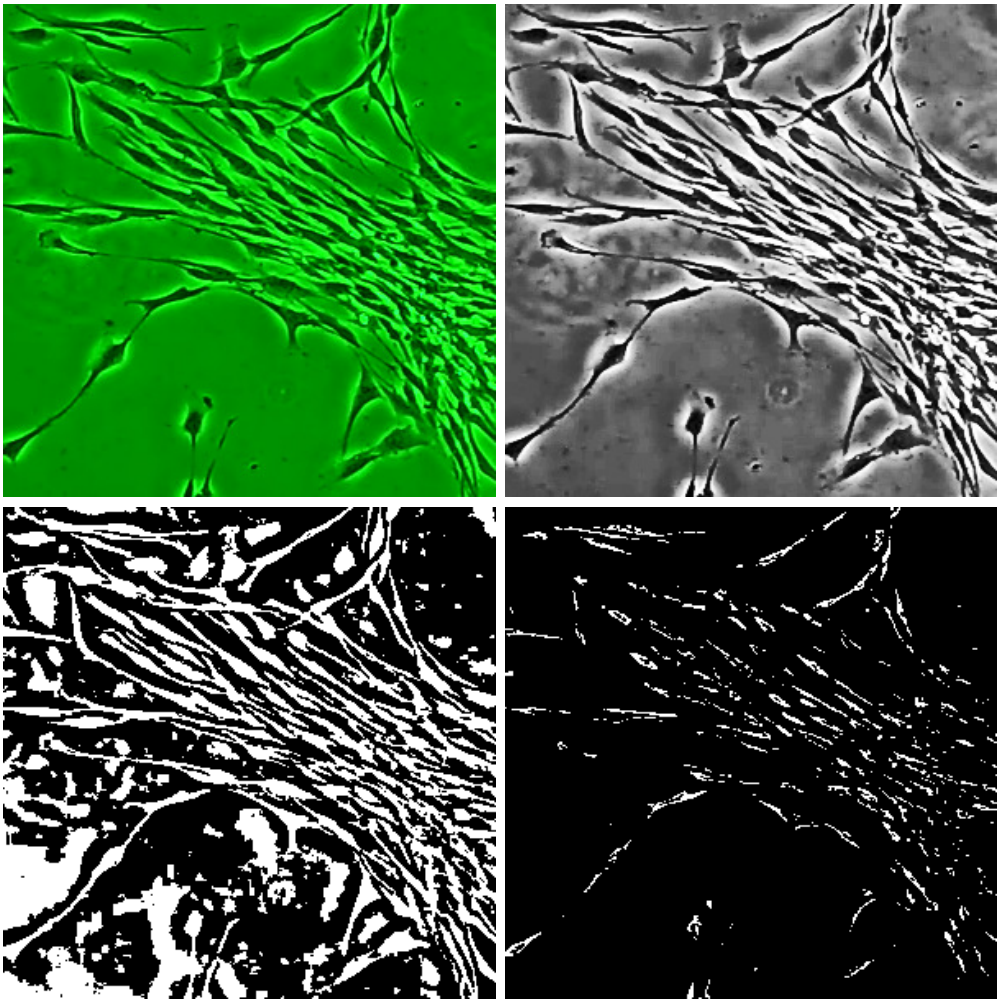
6.3.2 Rezultati

Primer preštetih fibroblastov je prikazan na sliki 6.10. Z oranžnimi križci so označene ročno preštete celice, z modrim pa celice, preštete z našim algoritmom. Vrednost optimalnih parametrov je prikazana v tabeli 6.6. Rezultati na testni množici so prikazani v tabeli 6.5. Glede na težavnost domene so tudi ti rezultati presenetljivo dobri. Naš števec z rastjo regij se je izredno dobro obnesel pri tem štetju. Na testni množici smo dosegli povprečno relativno napako 6,94 %, s standardnim odklonom 6,99 %. Na podlagi kumulativnega diagrama normalne porazdelitve (slika 6.11) lahko ugotovimo, da imamo kar 67-odstotno verjetnost, da na sliki naredimo relativno napako manjšo od 10 %.

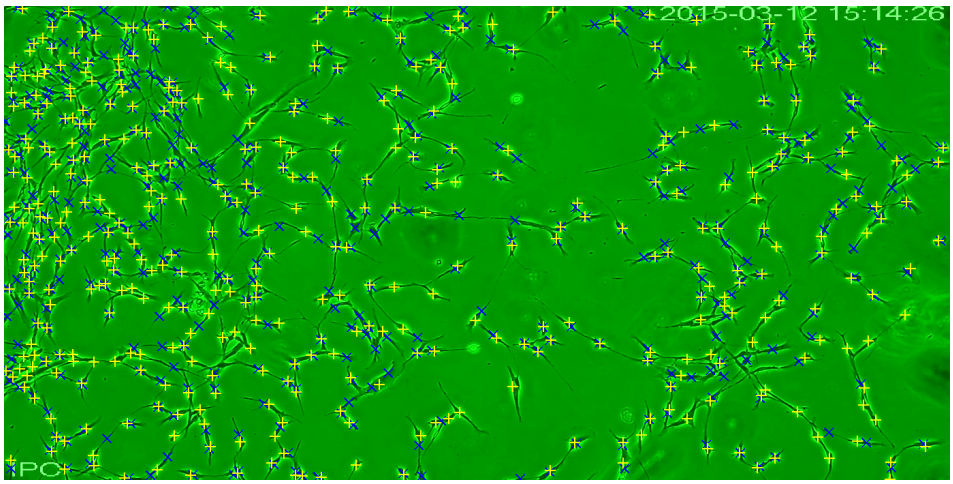
Kot je razvidno iz rezultatov, se je števec z rastjo regij dobro obnesel. Nekaj je k temu pripomoglo tudi dejstvo, da se pri tako velikem številu objektov

Algoritem 6: Psevdokoda štetja fibroblastov

```
1: <ObjectCounterRegionGrowing id="growing"
     sizeRange="1-2500" SeedThreshold ="13"
     MinCirc ="0.002286534222763814" MaxSize ="200"
     MaxCirc ="0.7584779311093457" MinSize ="55" MaxDiff ="45">
2:
3:   run("Gaussian Blur...",
       "sigma=<ParameterDouble id="gbsigma"
       minValue="0.0" maxValue="5.0"
       value ="0.43521119430655175" />");
4:
5:   openWindowName = getTitle()
6:   imgName = getTitle()
7:   blueTitle = imgName + "(blue)";
8:   redTitle = imgName + "(red)";
9:   greenTitle = imgName + "(green)";
10:
11:  run("Split Channels")
12:  selectWindow(redTitle);
13:  close()
14:  selectWindow(blueTitle);
15:  close()
16:  selectWindow(greenTitle)
17:
18:  run("Enhance Contrast...", "saturated=0 equalize");
19: </ObjectCounterRegionGrowing>
```



Slika 6.9: Binarizacija fitoblastov. Levo zgoraj: originalna slika, desno zgoraj: zeleni kanal, levo spodaj: binarizacija s pragom 85, desno spodaj: binarizacija s pragom 14.



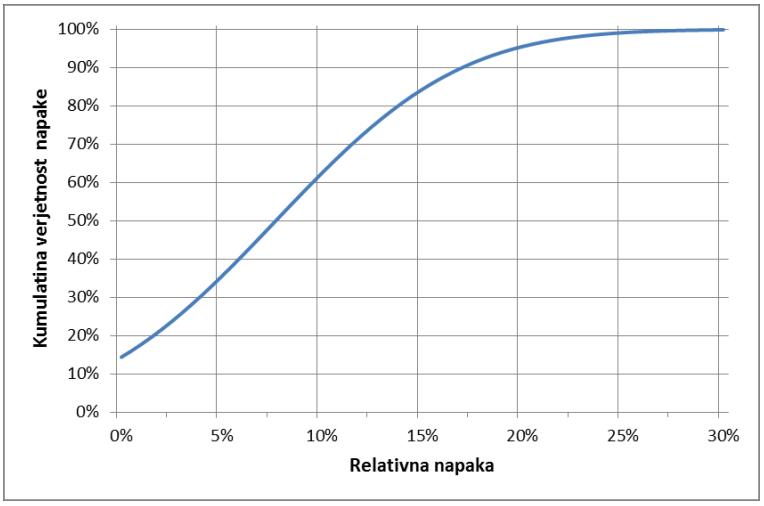
Slika 6.10: Rezultati štetja fibroblastov

Tabela 6.5: Rezultati štetja fibroblastov

#	Število objektov	Število preštetih objektov	Absolutna napaka	Relativna napaka (v %)
1	107	102	5	0,05
2	92	94	2	0,02
3	932	997	65	0,07
4	771	808	37	0,05
5	1316	1284	32	0,02
6	232	223	9	0,04
7	149	134	15	0,1
8	44	36	8	0,18
9	451	333	118	0,26
10	177	178	1	0,01
11	289	268	21	0,07
12	149	165	16	0,11
13	166	170	4	0,02

Tabela 6.6: Vrednost parametrov

Ime parametra	Vrednost parametra
growing:MinSize	55
growing:MaxSize	200
growing:MinCirc	≈ 0
growing:MaxCirc	$\approx 0,76$
growing:SeedThreashold	13
growing:MaxDiff	45
gbsigma	$\approx 0,44$



Slika 6.11: Kumulativni diagram napake števca fibroblastov

nekatere napake izničijo. Je pa pri tako zahtevnem štetju napaka odvisna tudi od natančnosti označevanja objektov. Ročno označevanje je težavno in zahteva veliko natančnost. Pri tem smo prepoznali slabost absolutne napake, saj imajo slike velik možen razpon števila celic in je absolutna napaka močno vplivala na uspešnost na slikah z veliko objekti. Z uvedbo relativne napake smo to oviro uspešno odstranili.

6.4 Teloh - reže tkiva listov

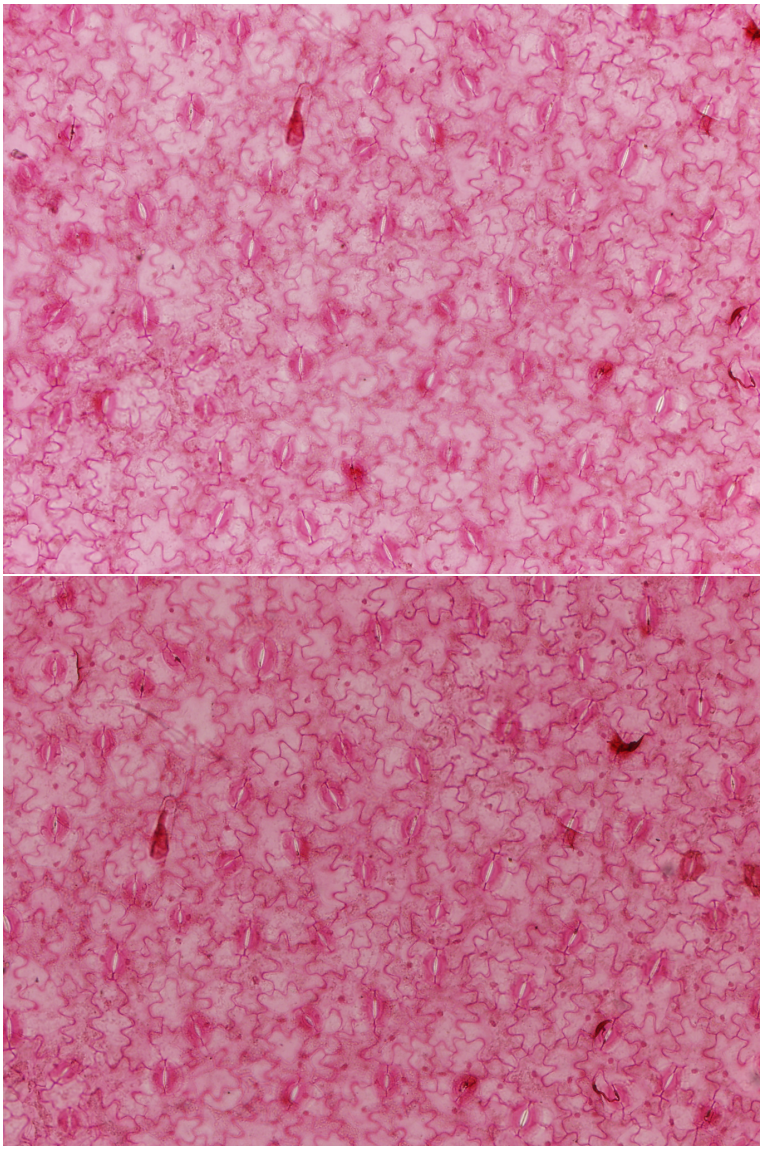
Iz Biotehniške fakultete smo dobili slike spodnjega površinskega tkiva lista teloha (slika 6.12). Tkivo so v fazi zajema obarvali z rdečim barvilom, s čimer so dosegli večji kontrast slike. Prešteti smo želeli reže na povrhnjici teloha. Reža je podolgovata svetla struktura (odprtina), ki jo obdajata dve upognjeni celici, in spominja na kavno zrno.

Število rež je pomemben ekološki znak, odvisno je od vlažnosti okolja, jakosti svetlobe in drugih dejavnikov. Število rež se lahko uporablja tudi kot merilo za razvrščanje in poimenovanje rastline (taksonomski znak). Znanstvenike večinoma zanima število rež na enoto površine (gostota). Za štetje uporabljajo programsko orodje cellSens od Olympusa [10], vendar ga niso primerno pripravili za štetje rež. Zato so slike posredovali nam, da jih poskusimo prešteti z našim pristopom.

6.4.1 Algoritem

Za štetje rež na spodnji povrhnjici teloha smo razvili števec, prikazan v algoritmu 7. Vhodne slike so tipa RGB. Zato smo tudi na tej domeni najprej poskusili posamezne kanale in njihove kombinacije. Da smo uravnali svetlosti čez celotno množico slik, smo najprej na vseh kanalih ločeno povečali kontrast. Kot kombinacija se je dobro obnesla razlika med zelenim in rdečim kanalom (od 3. do 13. vrstice algoritma). Pri tej kombinaciji smo dobro poudarili celici, ki obdajata režo.

Za še večji poudarek teh dveh celic smo na dobljeni sliki pognali metodo



Slika 6.12: Primeri slik

lokalnega povečanja kontrasta (15. vrstica). Lokalna povečava kontrasta se uporablja pri analizi slik. Metoda lokalne povečave kontrasta za prikaz slike ni priporočljiva, saj je rezultat nerealna slika. Ob tem tudi ni potrebna, saj se našim očem ni težko lokalno prilagoditi kontrastu in s slike razbrati objekte. Uporabili smo algoritem CLAHE (»Contrast Limited Adaptive Histogram Equalization«). Algoritem sliko razdeli v mrežo. Za vsako regijo se neodvisno izračuna histogram, katere vrednosti uporabi za razteg. Da se na končni sliki ne vidijo robovi slike, se za točko izračuna interpolirana vrednost njenih sosednih regij. Velikost regije določimo s parametrom »blocksize«. Število nivojev histograma podamo kot parameter »histogram«. Razteg histograma se izračuna s kumulativnim diagramom. S parametrom »max slope« omejimo maksimalni skok, ki ga kumulativni diagram lahko naredi. Če presežemo to vrednost, algoritem višek razporedi na druge nivoje histograma. S tem odstranimo težave s šumom, ki ga ima osnovna različica algorima (AHE). Vse tri parametre v našem algoritmu smo parametrizirali, da se ni bilo treba ukvarjati z natančnimi vrednostmi [11].

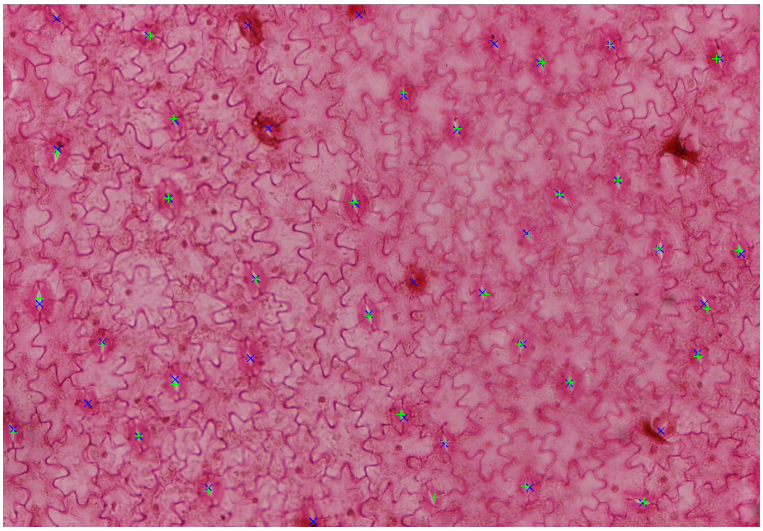
Nato smo na sliki pognali ukaz »Median...« (17. korak), ki je vsako slikovno piko nastavil na vrednost mediane njene okolice. Velikost okolice določa parameter »radius«, ki smo ga tudi parametrizirali. Nato smo v koraku 18. in 19. izvedli binarizacijo, pri čemer smo parametrizirali spodnjo vrednost praga.

V tem koraku naj bi dobili binarno sliko, na kateri sta dobro označeni celici, ki obdajata režo. Pred štetjem moramo ti celici nekako povezati v eno. To nam je uspelo z ukazi od 21. do 32. vrstice. Najprej smo v zanki ponavljali operacijo »Dilate«. Število ponovitev smo parametrizirali. Operacija »Dilate« povečuje in gladi regije. S tem smo želeli doseči, da se celici na obrobju rež povežeta. Opcijsko pokličemo ukaz »Fill Holes«, ki v maski zapolni luknje. Ali se ukaz izvede, določi genetski algoritem. V našem primeru bo ukaz zaprl morebitno režo med celicama. Nato v zanki ponavljamo operacijo »Erode«, ki je obratna operaciji »Dilate«. Nastalo sliko pošljemo osnovnemu števcu.

Algoritem 7: Psevdokoda štetja rež

```
<ObjectCounterAnalyzeParticles id="counter"
    MaxCirc="0.8842953629737577" MaxSize="99735"
    MinCirc="0.2822558051506735" MinSize="420">

    run("RGB Stack")
    run("Next Slice [&gt;]");
    run("Next Slice [&gt;]");
    run("Delete Slice")

    run("Enhance Contrast...",
      "saturated=0.3 normalize equalize process_all use");

    run("Stack to Images")
    run("Image Calculator...","image1=Green operation=Difference
      image2=Red create")
    close("Red");
    close("Green");

    run("CLAHE", "blocksize=<ParameterInt id="claheBlockSize"
      minValue="1" maxValue="500" value="326" /> histogram=
      <ParameterInt id="claheHistogram" minValue="1"
      maxValue="256" value="73"/> maximum=<ParameterInt
      id="claheMaximum" minValue="1"
      maxValue="1000" value="872"/>");

    run("Median...", "radius=<ParameterDouble id="filter1"
      minValue="0" maxValue="10" value="7.921553420043345" />");
    setThreshold(<ParameterInt id="thresholdMin" minValue="50"
      maxValue="255" value="226" />, 255);
    run("Convert to Mask");

```

```
21:     for (i=0; i &lt; <ParameterInt id="iterationsDilade"
          minValue="0" maxValue="30" value="8" />; i++) {
22:       run("Dilate");
23:     }
24:     <optional id="optionalFillHoles" value="true" IsEnabled ="true">
25:       run("Fill Holes");
26:     </optional>
27:     for (i=0; i &lt; <ParameterInt id="iterationsErode"
          minValue="0" maxValue="30" value="11" />; i++) {
28:       run("Erode");
29:     }
30:     for (i=0; i &lt; <ParameterInt id="iterationsMedian"
          minValue="0" maxValue="30" value="29" />; i++) {
31:       run("Median...", "radius=<ParameterDouble id="filter2"
            minValue="0" maxValue="10"
            value="1.151253239976897" />");
32:     }
33: </ObjectCounterAnalyzeParticles >
```



Slika 6.13: Rezultati štetja rež

6.4.2 Rezultati

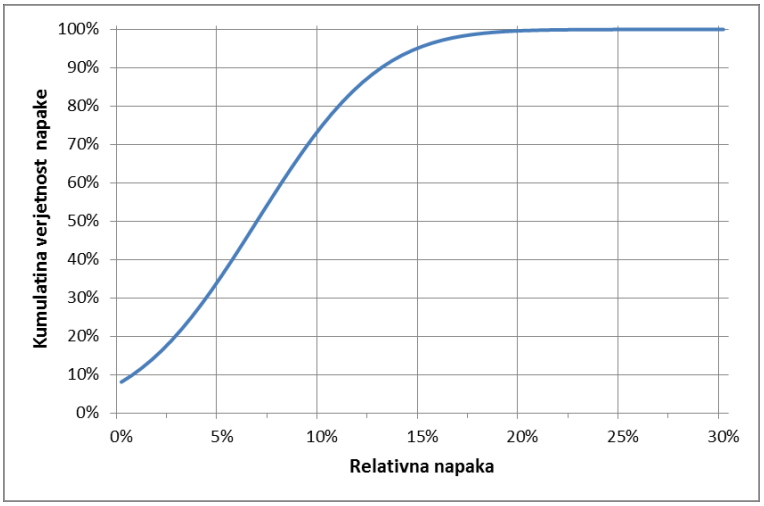
Primer preštetih rež je prikazan na sliki 6.13. Z zelenim križcem so označene ročno preštete celice, z modrim pa celice, preštete z našim algoritmom. Vrednost optimalnih parametrov je prikazana v tabeli 6.8. Rezultati na testni množici so prikazani v tabeli 6.7. Na testni množici smo dosegli povprečno relativno napako 6,75 %, s standardnim odklonom 4,83 %. Na podlagi kumulativnega diagrama normalne porazdelitve (slika 6.14) lahko ugotovimo, da imamo kar 79-odstotno verjetnost, da na sliki naredimo relativno napako, manjšo od 10 %.

Tabela 6.7: Rezultati štetja rež teloha

#	Število objektov	Število preštetih objektov	Absolutna napaka	Relativna napaka (v %)
1	51	52	1	0,02
2	48	49	1	0,02
3	47	52	5	0,11
4	44	48	4	0,09
5	57	58	1	0,02
6	60	51	9	0,15
7	53	49	4	0,08
8	44	42	2	0,05

Tabela 6.8: Vrednost parametrov

Ime parametra	Vrednost parametra
counter:MaxSize	99735
counter:MinSize	420
counter:MaxCirc	$\approx 0,88$
counter:MinCirc	$\approx 0,28$
claheBlockSize	326
claheHistogram	73
claheMaximum	872
filter1	$\approx 7,92$
thresholdMin	226
iterationsDilade	8
optionalFillHoles:IsEnabled	true
iterationsErode	11
iterationsMedian	29
filter2	$\approx 1,15$



Slika 6.14: Kumulativni diagram štetja rež

6.5 Štetje ptic v jati

Domene mikroskopskih slik so dokaj predvidljive, saj so bile posnete v nadzorovanih okoljih. To se je dobro videlo pri prejšnjih preizkusih, saj smo pri vseh domenah imeli majhen standardni odklon relativne napake. Kot eno testno domeno smo želeli množico slik, ki ne bi bila posneta z mikroskopom. Za štetje objektov na takih domenah so na splošno uporabnejše razne statistične metode, ki objekte izračunajo na podlagi primerjave značilk [12]. Za našo nemikroskopsko domeno smo izbrali slike ptic (slika 6.15). Poskusili bomo sestaviti števec, ki bo uspešno preštel število ptic v jati.

6.5.1 Algoritem

Števec za štetje ptic v jati je prikazan v algoritmu 8. Pričakujemo RGB-slike, saj gre za slike, ki so posnete s klasičnimi digitalnimi fotoaparati. Najboljša kombinacija kanalov je bila ta, da smo sliko najprej pretvorili iz RGB-barvnega prostora v HSB-barvni prostor. HSB pomeni barvni odtenek (»hue«), nasičenost (»saturation«) in svetlost (»brightness«). V HSB-barvnem prostoru smo uporabili kanal svetlosti. Postopek je prikazan od 2.

Slika 6.15: Primeri slik

do 4. vrstice algoritma.

Na dobljenih slikah imamo zdaj temne ptice s svetlim ozadjem. Ker so slike posnete v različnih svetlobnih pogojih, bomo poskusili na vseh slikah ustvariti čim bolj belo ozadje. Ker predpostavljamo, da je večji del slike ozadje, bomo za povprečno svetlost ozadja vzeli kar povprečno svetlost slike. Celotni sliki za vsak slikovni element prištejemo svetlost ozadja, pomnoženo s skalarjem (6.1). Nastavljanje vrednosti skalarja bomo prepustili optimizacijskemu algoritmu. Ta postopek je v 5. in 6. vrstici algoritma. Po tem izvedenem ukazu na vseh slikah pričakujemo popolnoma belo ozadje. To nam bo omogočilo lažje določanje praga binarizacije v nadaljevanju.

$$nova\_intenziteta = prejsnja\_intenziteta + povprecna\_intenziteta * skalar \tag{6.1}$$

Sledila je povečava kontrasta (7. vrstica). Nato smo sliko binarizirali (8. in 9. vrstici algoritma). Zgornjo pragovno vrednost smo pri tem parametrizirali. Tudi pri tem smo nato v zanki ponavljali operacijo »Dilate«, ki povečuje regije. Tudi število ponovitev smo parametrizirali. Dobljeno sliko je nato preštel števec objektov.

6.5.2 Rezultati

Primer preštetih ptic v jati je prikazan na sliki 6.16. Z rumenim križcem so označene ročno preštete ptice, z modrim pa ptice, preštete z algoritmom. Vrednost optimalnih parametrov je prikazana v tabeli 6.10. Rezultati na testni množici so prikazani v tabeli 6.9. Kot je razvidno iz tabele, se dobljeni rezultati izredno razlikujejo od slike do slike. To je bilo tudi pričakovati, saj so slike med seboj zelo različne. Naša povprečna relativna napaka je 20,8 %, s standardnim odklonom 18,3 %. Na podlagi kumulativnega diagrama normalne porazdelitve (slika 6.17) lahko ugotovimo, da imamo le 28-odstotno možnost, da na sliki naredimo relativno napako, manjšo od 10 %. Zaradi velikega standardnega odklona se tudi za relativno napako, manjšo od 20 %, verjetnost bistveno ne izboljša. Za manjšo ali enako kot 20 % imamo le

Algoritem 8: Psevdokoda štetja ptic

```
1: <ObjectCounterAnalyzeParticles id="objectCounter"
       MinCirc ="0.03095819498020247" MaxSize ="13841"
       MinSize ="138" MaxCirc ="0.934169321571594" >
2:     run("HSB Stack");
3:     run("Delete Slice");
4:     run("Delete Slice");
5:     getStatistics(area, mean, min, max, std, histogram)
6:     run("Add...", "value="+ (255 - mean) *
         <ParameterDouble id="meanMulti" minValue="0.0"
         maxValue="5.0" value="0.5132035277140073" />);
7:     run("Enhance Contrast...", "saturated=0.3 normalize equalize");
8:     setThreshold(0, <ParameterInt id="threasholdMax"
         minValue="0" maxValue="255" value="43" />);
9:     run("Convert to Mask");
10:    for (i=0; i &lt; <ParameterInt id="iterations" minValue="0"
         maxValue="30" value ="25" />; i++) {
11:      run("Dilate");
12:    }
13: </ObjectCounterAnalyzeParticles >
```

Slika 6.16: Rezultati štetja ptic

48-odstotno možnost.

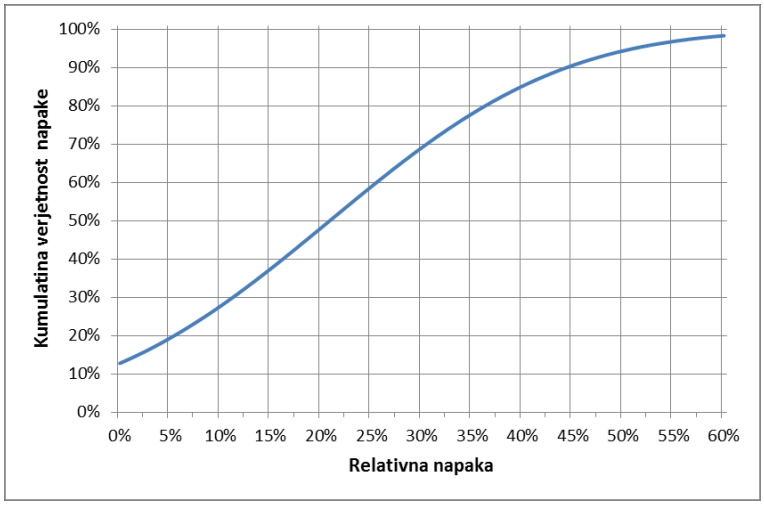
Na tej množici slik imamo še veliko možnosti, kako posamezno sliko prešteti bolje, s tem pa bi pokvarili rezultate na drugih slikah. Hoteli smo razviti čim bolj robusten algoritem, ki bi bil primeren za čim večji spekter slik jat. Korake odstranjevanja ozadja bi lahko preprosto nadomestili z vgrajenim ukazom ImageJ »Subtract Background...«. Ukaz je na večini slik prinesel izredno dobre rezultate odstranjevanja ozadja. Pri nekaterih slikah pa se je izkazal zelo slabo. Pri slikah, na katerih so bile ptice velike, je tudi ptice zaznal kot ozadje. Zato smo ga morali nadomestili z našim robustnejšim in preprostejšim načinom odstranjevanja ozadja.

Tabela 6.9: Rezultati štetja ptic

#	Število objektov	Število preštetih objektov	Absolutna napaka	Relativna napaka (v %)
1	120	85	35	0,29
2	26	21	5	0,19
3	50	38	12	0,24
4	39	38	1	0,03
5	26	39	13	0,5
6	15	15	0	0

Tabela 6.10: Vrednost parametrov

Ime parametra	Vrednost parametra
objectCounter:MaxSize	13841
objectCounter:MinSize	138
objectCounter:MaxCirc	$\approx 0,93$
objectCounter:MinCirc	$\approx 0,03$
meanMulti	$\approx 0,51$
threasholdMax	43
iterations	25



Slika 6.17: Kumulativni diagram štetja ptic

Poglavje 7

Sklepne ugotovitve

V magistrski nalogi smo za podani števec poskušali poiskati nastavitve, ki bi čim bolje preštele iskane objekte na sliki. Kot osnovo za opis algoritma smo uporabili makro jezik ImageJ. Jezik smo morali za naše potrebe nekoliko razširiti. Za podporo razširitvam smo zasnovali čim bolj splošno razširjeni makro jezik ImageJ. Sintaksa jezika omogoča preprosto nadaljnje dodajanje in spreminjanje razširitev. Jezik smo razširili z dvema števcema. Naš prvi števec, imenovan števec delcev, na binarni sliki prešteje objekte, ki ustrezajo omejitvam velikosti in kompaktnosti. Naš drugi števec, imenovan števec delcev z rastjo področij, deluje podobno kot prvi, le da v primerjavi s prvim na začetek dodamo korak, v katerem izvajamo rast regij. V jezik smo dodali parametre, s katerimi fiksne vrednosti parametriziramo. Podprli smo parametre s celoštevilsko, decimalno in logično vrednostjo. Parametrom v fazi optimizacije z genetskim algoritmom poskušamo čim bolje nastaviti vrednost. Razvili smo tudi uporabniški vmesnik, ki olajša delo strokovnjakom, ki razvijajo števec.

Z našo aplikacijo uporabnikom omogočimo lažje izvajanje samodejnega štetja objektov na sliki. S tem jih razbremenimo mučnega in zamudnega štetja, ki so ga zdaj opravljali ročno. Optimizacija tudi poskrbi, da iskanje števca ne bo samo lažje, ampak praviloma tudi točnejše.

7.1 Nadaljnje delo

Naša pozornost je bila večinoma namenjena kakovosti iskanja optimalnega števca, in ne toliko hitrosti iskanja. Na področju hitrosti izvajanja smo poskrbeli za paralelizacijo, ki na sodobnih sistemih prihrani ogromno časa. Gre za težavo z veliko stopnjo paralelizma (ločeno poganjamo več konfiguracij parametrov na več slikah), zato imamo zelo dober izkoristek parelelizacije. Med iskanjem najboljše rešitve smo preizkusili veliko kombinacij vrednosti parametrov. Za veliko kombinacij bi lahko že pred izvajanjem ugotovili, da jih ni smiselno poganjati. Ni smiselno poganjati nastavitev, pri katerih je na primer spodnji prag binarizacije večji kot zgornji prag binarizacije. Prav tako nima smisla poganjati nastavitev, pri katerih je najmanjša dovoljena velikost objekta večja od največje dovoljene velikosti. Za take primere bi bilo koristno v nadaljnjem delu uvesti možnost, da med parametri lahko vzpostavimo dodatne omejitve. Tako bi na primer določili, da mora biti spodnji prag binarizacije manjši od zgornjega praga. Da neki osebek sprejmemo v populacijo, morajo na njem veljati vse omejitve. Tako bi lahko že samo pri gradnikih tipa števec močno zmanjšali prostor vseh možnih stanj.

Naša želja je, da končne uporabnike čim manj omejujemo pri uporabi števca. Zato je želja v prihodnosti podpreti možnost, da po končani optimizaciji števec pretvorimo v osnovni makro jezik. Trenutno tega ne moremo narediti, saj ta konverzija ni na voljo za oba števca. Definicijo števcev bi morali v ta namen nekoliko posplošiti. Trenutna zasnova je bila zastavljena z mislijo, da bomo paralelizacijo izvedli z nitenjem v okviru enega procesa. Pozneje se je izkazalo, da zasnova knjižnice ImageJ to onemogoča. Interpreter ImageJ uporablja globalne spremenljivke, ki onemogočajo sočasno izvajanje več makrojev. Dobili bi nepredvidljive rezultate. Ko smo ovrgli možnost paralelizacije z nitenjem, se ponovne implementacije števcev nismo lotevali.

Za uporabo optimizacije smo realizirali ločen uporabniški vmesnik. Smiselno bi bilo v nadaljevanju razmisliti o možnosti, da bi uporabniški vmesnik integrirali v orodje ImageJ. Tako bi podobno funkcionalnost, kot jo ima zdaj naš uporabniški vmesnik, zapakirali v vtičnik ImageJ. Popolna integracija

naše rešitve v vtičnik ImageJ bi še dodatno pripomogla k uporabnosti in prijaznosti samodejnega iskanja števcev objektov na slikah.

Literatura

[1] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: Principles and Practice of Constraint Programming-CP 2009, Springer, 2009, pp. 142–157.

[2] D. S. Bolme, J. R. Beveridge, B. A. Draper, P. J. Phillips, Y. M. Lui, Automatically searching for optimal parameter settings using a genetic algorithm, in: Computer Vision Systems, Springer, 2011, pp. 213–222.

[3] Rasband, W.S., ImageJ, U. S. National Institutes of Health, Bethesda, Maryland, USA, `http://imagej.nih.gov/ij/`, 1997-2015.

[4] Seznam vtičnikov ImageJ, dosegljiv na `http://rsb.info.nih.gov/ij/plugins/` (marec 2016).

[5] R. Adams, L. Bischof, Seeded region growing, Pattern Analysis and Machine Intelligence, IEEE Transactions on 16 (6) (1994) 641–647.

[6] Meffert, Klaus, JGAP - Java Genetic Algorithms and Genetic Programming Package, `http://jgap.sf.net`, datum dostopa: 2015-02-15.

[7] B. Yuan, M. Gallagher, A hybrid approach to parameter tuning in genetic algorithms, in: Evolutionary Computation, 2005. The 2005 IEEE Congress on, Vol. 2, IEEE, 2005, pp. 1096–1103.

[8] J. Lojk, U. Čibej, D. Karlaš, L. Šajn, M. Pavlin, Comparison of two automatic cell-counting solutions for fluorescent microscopic images, Journal of microscopy 260 (1) (2015) 107–116.

[9] Fotona XD-2, dosegljivo na `http://www.fotona.com/es/products/?id=257` (marec 2016).

[10] Orodje Olympus cellSens, dosegljivo na `http://www.olympus-lifescience.com/en/software/cellsens/` (marec 2016).

[11] K. Zuiderveld, Graphics gems iv, Academic Press Professional, Inc., San Diego, CA, USA, 1994, Ch. Contrast Limited Adaptive Histogram Equalization, pp. 474–485.
URL `http://dl.acm.org/citation.cfm?id=180895.180940`

[12] M. Ranzato, P. Taylor, J. House, R. Flagan, Y. LeCun, P. Perona, Automatic recognition of biological particles in microscopic images, Pattern Recognition Letters 28 (1) (2007) 31–39.